Gravitas: An extensible physics engine framework using object-oriented and design pattern-driven software architecture principles

Colin Vella

Supervisor: Dr. Ing. Adrian Muscat

Department of Communications and

Computer Engineering

September 2008

Submitted in partial fulfilment of the requirements

for the degree of Master in I.T.

Abstract

We present a physics engine architecture specifically designed to support a wide gamut of rigid body simulation techniques, algorithms and features via abstraction layers and a modular plug-in mechanism. We corroborate our design through the implementation of the engine framework and a number of plug-ins to enable physics simulations featuring a wide variety of body geometry, spatial representations, forces, constraints and collision and constraint resolution techniques. The engine does not impose a fixed simulation pipeline, and hence it is possible to implement many simulation approaches, ranging from the simple impulse-based, retroactive techniques to more complex approaches involving conservative advancement, continuous collision detection and simultaneous constraint resolution. We thus demonstrate that it is practical to architect a generic physics engine framework operating at real-time interactive rates on commodity computer hardware. We also propose an abstract motion constraint specification and present a number of joint models based upon it, which we subsequently demonstrate by means of their software implementation and the derivative results.

Acknowledgement

This thesis would not have been possible without the support of many people. I thank my supervisor Dr. Adrian Muscat for his ongoing guidance and support. I thank my close friends and colleagues at Crimsonwing for the pleasure of their company and the wisdom of their advice. Finally I thank my natural and my extended family for their support, and especially my wife Claire, for taking on much of the burden related to the running of our day-to-day life and the care of our daughter Tamara.

Dedication

I dedicate this work to my dear daughter Tamara, for giving me the opportunity to experience the joys and toils of fatherhood.

Statement of Authenticity

Table of Contents

Chapter 1 Introduction ... 1
1.1 Prelude ... 1
1.2 Physics Modelling Approaches ... 2
1.2.1 Analytically-Based Modelling ... 2
1.2.2 Numerically-Based Modelling ... 2
1.3 Aggregate Mass Systems and Soft Body Modelling ... 3
1.4 Collision Detection ... 3
1.5 Collision Response ... 5
1.6 Joints and Actuators ... 7
1.7 Scaling Considerations ... 8
Chapter 2 Background ... 9
2.1 Physics Engines ... 9
2.2 Spatial Transformations ... 9
2.2.1 Homogenous Matrix Transforms ... 9
2.2.2 Quaternion-Based Transforms ... 11
2.3 Kinetics ... 15
2.3.1 Linear Motion ... 15
2.3.1.1 Forces and Mass ... 15
2.3.1.2 Linear Kinetic Equations of Motion ... 16
2.3.2 Angular Motion ... 16
2.3.2.1 Centre of Mass ... 17
2.3.2.2 Torque and Moment of Inertia ... 18
2.3.2.3 Common Inertia Tensor Results ... 21
2.3.2.4 Angular Kinetic Equations of Motion ... 23
2.4 Numerical Integration ... 24
2.4.1 Euler Method ... 24
2.4.2 Verlet Methods ... 25
2.4.3 Fourth-Order Runge-Kutta Method ... 26
2.5 Collision Detection ... 26
2.5.1 Intersection Tests ... 27
2.5.1.1 The method of Separating Axes ... 29
2.5.1.2 The Gilbert-Johnson-Keerthi Algorithm ... 30
2.5.2 Time of Impact Estimation ... 31
2.5.2.1 Sphere-to-Sphere Test ... 32
2.5.2.2 Sphere-to-Half-Space Test ... 33
2.5.2.3 Tests for more Complex Geometry ... 34
2.5.3 Contact Manifold Generation ... 34
2.5.3.1 Reduced Contact Manifold ... 34
2.5.3.2 Manifold for Sphere-to-Sphere Contact ... 36
2.5.3.3 Manifold for Box-to-Half-space Contact ... 37
2.5.3.4 Manifold Generation for more Complex Geometry ... 38
2.6 Collision Response ... 39
2.6.1 Contact Forces ... 39
2.6.1.1 Reaction ... 39
2.6.1.2 Surface-to-Surface Friction ... 40
2.6.2 Impulse-based Contact Model ... 40

2.6.2.1 Impulse-based Reaction ... 41
2.6.2.1 Impulse-based Friction ... 43
2.6.3 Penalty-based Contact Model ... 44
2.6.3.1 Penalty-based Reaction ... 44
2.6.3.2 Penalty-based Friction ... 45
2.6.4 Rolling and Spinning Friction ... 47
2.6.5 Surface Imperfections ... 49
2.6.6 Local Collision Resolution ... 50
2.6.7 Simultaneous Collision Resolution ... 51
2.6.7.1 Iterative Methods ... 51
2.6.7.2 Linear Complementary Problem-based Formulations ... 52
2.6.7.3 Block Solvers ... 53
2.7 Constraints ... 55
2.7.1 Lagrangian-based Dynamics ... 56
2.7.2 Newtonian Dynamics ... 56
2.7.3 Unified Constraint Models ... 56
2.7.3.1 Jacobian Matrix Formulation ... 57
2.7.3.2 Jacobian Constraint Solvers ... 58
2.7.4 An Abstract Unified Constraint Model ... 59
2.7.5 Constraint Case Studies ... 60
2.7.5.1 Collision and Contact ... 60
2.7.5.2 Ball-and-Socket Joint ... 61
2.7.5.3 Hinge Joint ... 62
2.7.5.4 Rigid Joint ... 62
2.7.5.5 Angular Motor Joint ... 63
2.7.5.6 Joint Limits ... 64
2.7.5.7 Breakable Joints ... 65
2.7.6 Constraint Solvers ... 66

Chapter 3 Engine Design and Implementation ... 68
3.1 Design Principles ... 68
3.1.1 Reusability and Flexibility ... 68
3.1.2 Performance ... 68
3.1.3 Robustness ... 69
3.1.4 Plausibility ... 69
3.1.5 Visibility and Modularity ... 70
3.1.6 Abstraction and Extensibility ... 70
3.1.7 Consistency ... 70
3.2 Gravitas Organisational Structure ... 71
3.3 Foundations ... 72
3.3.1 Common Library ... 72
3.3.2 Gravitas Base Class ... 72
3.4 Collision Manager ... 73
3.4.1 Collision Detectors ... 73
3.4.1.1 Intersection Tests ... 73
3.4.1.2 Time of Impact Tests ... 74
3.4.2 Contact Manifold Generation ... 74
3.4.3 Collision Filter ... 75
3.4.4 Collision Groups ... 76

3.5 Constraint Manager 76
3.5.1 Constraints 76
3.5.2 Constraint Solvers 77
3.5.3 Constraint Batches 78
3.6 Event Manager 78
3.6.1 Activity Events 79
3.6.2 Collision Events 79
3.6.3 Collision Filtering Events 80
3.6.4 Constraint Events 80
3.6.5 Material Events 80
3.7 Force Manager 81
3.7.1 Forces 81
3.7.2 Force Accumulators 81
3.8 Geometry Manager 82
3.8.1 Body Geometry 82
3.8.2 Spatial Transform 83
3.8.3 Bounding Volumes 83
3.8.4 Geometric Primitives 84
3.8.4.1 Line Segments 84
3.8.4.2 Rays 84
3.8.4.3 Triangles 84
3.8.4.4 Tetrahedrons 85
3.9 Instrumentation Manager 85
3.10 Kinetics Manager 86
3.10.1 Kinetic Integrators 86
3.10.2 Kinetic Properties 87
3.10.3 Vector Time Functors 87
3.10.4 Activity Management 87
3.11 Material Manager 87
3.11.1 Materials 88
3.11.2 Material Mixing 88
3.12 Simulation Manager 89
3.12.1 Simulators 89
3.12.2 Environments 89
3.13 Space Manager 90
3.13.1 Candidate Collision Lists 91
3.13.2 Ray Tests 91
3.14 Miscellaneous Entities 92
3.14.1 Bodies 92
3.14.2 Mass Properties 92
Chapter 4 Plug-in and Application Case Studies 93
4.1 Core Constraints and Constraint Solvers Plug-in 93
4.1.1 Constraint Implementations 93
4.1.2 Constraint Solver Implementations 93
4.2 Core Forces Plug-in 93
4.3 Core Geometry and Collision Detectors Plug-in 94
4.3.1 Geometry Implementations 94
4.3.1.1 Spheres 94

4.3.1.2 Boxes....95
4.3.1.3 Cylinders....95
4.3.1.4 Convex Polyhedra....95
4.3.1.5 Triangle Meshes....95
4.3.2 Collision Detector Implementations....96
4.3.2.1 Sphere with Sphere....96
4.3.2.2 Box with Sphere....96
4.3.2.3 Box with Box....97
4.3.2.4 Cylinder with Sphere....97
4.3.2.5 Cylinder with Box....97
4.3.2.6 Cylinder with Cylinder....97
4.3.2.7 Convex Polyhedron with Sphere....98
4.3.2.8 Convex Polyhedron with Box....98
4.3.2.9 Convex Polyhedron with Cylinder....98
4.3.2.10 Convex Polyhedron with Convex Polyhedron....98
4.3.2.11 Triangle Mesh with Sphere....98
4.3.2.12 Triangle Mesh with Box....99
4.3.2.13 Triangle Mesh with Cylinder....99
4.3.2.14 Triangle Mesh with Convex Polyhedron....99
4.3.2.15 Half-Space with Sphere....99
4.3.2.16 Half-Space with Box....99
4.3.2.17 Half-Space with Cylinder....100
4.3.2.18 Half-Space with Convex Polyhedron....100
4.3.2.19 Half-Space with Triangle Mesh....100
4.4 Core Integrators Plug-in....100
4.5 Core Simulators Plug-in....101
4.6 Core Spaces Plug-in....101
4.7 Gravitas Sandbox Application....102

Chapter 5 Evaluation....103

Chapter 6 Conclusions and Future Work....109
6.1 Conclusions....109
6.2 Future Work....109
6.2.1 Constraint Model Validation....109
6.2.2 Constraint Solver Implementations....109
6.2.3 Force Implementations....110
6.2.4 Estimated Time of Impact Query Implementations....110
6.2.5 Simulation Pipeline Implementations....110
6.2.6 Robust implementation of Triangle Mesh Collisions....110
6.2.7 Enhanced Sandbox Application....111
6.2.8 Soft Body Dynamics....111

Table of Figures

Figure 1 Physics-based games from left to right: Armadillo Run; Gish; Switchball 1
Figure 2 Approximating complex visual geometry by simple physical geometry 4
Figure 3 From left to right: on collision course; interpenetration; exact contact 4
Figure 4 A 2D representation of an octree 5
Figure 5 Bounding volumes, from left to right: no collision; potential collision; definite collision 5
Figure 6 Parameters of interest in a simulated collision between two bodies 6
Figure 7 A typical configuration of bodies stacked on a ground plane 6
Figure 8 Common joint types, clockwise from top left: ball-and-socket; hinge; CV joint; spring; slider; axle 7
Figure 9 Simulating organic joints using skinning 8
Figure 10 The sleep/wake algorithm in Bullet demonstrated using colour-coded bodies..........8
Figure 11 Unit quaternion represented as a point on the unit 4D hypersphere 12
Figure 12 Regular point sampling for approximate mass property computation 18
Figure 13 Moments of inertia for a sphere around the principal axes 21
Figure 14 Moments of inertia for a box around the principal axes 22
Figure 15 Moments of inertia for a cylinder around the principal axes 22
Figure 16 Euler method approximation of a function integral 25
Figure 17 Predicate function for intersection testing 27
Figure 18 Sphere-to-sphere intersection testing 27
Figure 19 Box-to-sphere intersection testing 29
Figure 20 The method of separating axes..29
Figure 21 Minkowski difference of two convex regions..30
Figure 22 TOI estimation between two spheres 32
Figure 23 TOI estimation between a sphere and a half-space 33
Figure 24 Contact manifolds for various geometry and contact configurations 34
Figure 25 Contact manifold reduction by the principle of contact force and impulse equivalence 35
Figure 26 The six geometrical feature contact cases 35
Figure 27 Two spheres in contact...37
Figure 28 A box in contact with a half-space 38
Figure 29 The compression and expansion phases of collision..40
Figure 30 Friction due to surface microstructure imperfections 40

Figure 31 The application of impulses at the point of collision 41
Figure 32 Friction cone resulting from the Coulomb friction model 43
Figure 33 The application of damped springs to penalise body interpenetration............ 45
Figure 34 Static friction penalty force............ 46
Figure 35 Dynamic friction penalty force 47
Figure 36 Body in a rolling configuration............ 47
Figure 37 Extended contact manifolds to induce rolling friction............ 48
Figure 38 Using polyhedral geometry approximations to induce rolling friction............ 49
Figure 39 Varying contact normals due to surface microstructure imperfections 49
Figure 40 Perturbing a surface normal within a spherical cap 49
Figure 41 Solving interpenetration by linear translation............ 50
Figure 42 Limitations of a local collision resolver............ 51
Figure 43 Partitioning a simulation into islands of bodies in contact 54
Figure 44 Articulated body amenable to the Featherstone method............ 56
Figure 45 Discrepancy-based constraint model for collision............ 61
Figure 46 Discrepancy-based constraint model for a ball-and-socket joint............ 61
Figure 47 Discrepancy-based constraint model for a hinge joint............ 62
Figure 48 Discrepancy-based constraint model for a rigid joint 63
Figure 49 Discrepancy-based constraint model for an angular motor joint 64
Figure 50 Discrepancy-based constraint model of a limited joint............ 65
Figure 51 The Meson simulation platform............ 71
Figure 52 High-level view of the Gravitas engine 71
Figure 53 Collision manager 73
Figure 54 Intersection testing 74
Figure 55 Time of impact estimation 74
Figure 56 Contact manifold generation............ 75
Figure 57 Collision filtering 76
Figure 58 Constraint management............ 77
Figure 59 Constraint solver management............ 77
Figure 60 Constraint batching 78
Figure 61 An example of the event manager servicing another module............ 79
Figure 62 Force manager............ 81
Figure 63 Force accumulators 82
Figure 64 Geometry manager............ 82

Figure 65 Instrumentation manager..85
Figure 66 Kinetics manager..86
Figure 67 Material manager 88
Figure 68 Simulation manager 89
Figure 69 Simulation environment..90
Figure 70 Space manager 90
Figure 71 Modular organisation of the Gravitas Sandbox Application.......................102
Figure 72 A stacked configuration of boxes leveraged by a counter weight 104
Figure 73 A simulation of the domino effect 104
Figure 74 A demonstration of Newton's Cradle 105
Figure 75 Force implementation example illustrating bodies caught in a simulated vortex..105
Figure 76 Simulation with instrumentation enabled 106
Figure 77 A ball running down a triangle-mesh track (17784 vertices, 35564 faces) 106
Figure 78 A simulation of stacked and articulated bodies...107
Figure 79 Ragdoll physics using spherical and revolute joints 107
Figure 80 A spinning top exhibiting precession...108
Figure 81 A simulated mechanical quadruped 108

Chapter 1
Introduction

In this chapter we present a review of rigid body dynamics simulation with a focus on real-time systems employed in interactive simulations and computer games. In Chapter 2 we survey the theory and mathematics pertaining to physics modelling, with the purpose of identifying generalisations or recurring aspects of the relevant concepts. Based on the insights gained from Chapter 1 and Chapter 2, we describe, in Chapter 3, a software architecture for a physics engine framework, titled *Meson: Gravitas*, that encourages a wide range of simulation techniques and features, based on object oriented paradigms [**1**] and design patterns [**2**]. In Chapter 4 we present a number of implemented components, or *plug-ins*, showcasing the flexibility and effectiveness of the physics engine framework, concluding with a description of the Gravitas Sandbox application based on the physics engine and the Vistas visualisation engine [**3**], which was developed to allow users to experiment with the engine and its various plug-in components. In Chapter 5 we evaluate the work based on the achieved results and discuss potential approaches for establishing the quality of the results. Finally, in Chapter 6 we draw our conclusions with respect to the objectives set out for this dissertation, and highlight potential areas of related further work.

1.1 Prelude

Modern software applications relating to science, engineering and the entertainment media often require the ability to simulate physical effects, in particular, rigid and soft body dynamics, termed simply as *physics* in this context. Typical applications include forecasting results in engineering design software, creating special effects using computer generated imagery (CGI) in movies and providing a high degree of player immersion through realistic interactive environments within computer games. In some games, such as Armadillo Run [**4**], Gish [**5**] and Switchball [**6**], the game play focuses entirely around physics (Figure 1), where the player is required to solve physical puzzles to progress through the games.

Figure 1 Physics-based games from left to right: Armadillo Run; Gish; Switchball

1.2 Physics Modelling Approaches

The simulation of rigid and soft body dynamics, which we term simply as *physics*, may be broadly categorised into analytically-based and numerically-based modelling. This work will focus on the more generic numerically-based approaches. However, we briefly discuss analytically-based modelling to highlight its differences with respect to numerical methods.

1.2.1 Analytically-Based Modelling

The study of analytically-based physics modelling, often termed *Lagrangian dynamics* [**7**], concerns itself with the derivation of equations that completely describe the motion of a system of bodies using a generalised coordinate system, and as such, incorporate all the acting agents and relevant constraints, including non-penetration, contact forces, friction and joints.

This approach requires mathematical analysis of the system in question, followed by a specific software implementation for the system, or at least, for the family of similar systems. The chaotic nature of such systems, often describable only in terms of systems of differential equations, limits this approach to relatively simple configurations.

Analytical models tend to be well suited in systems where the motion of bodies is constrained and predictable, but are not appropriate in situations that involve dynamic interactions between multiple bodies. Typical examples of suitable applications include projectiles, pendulums, damped springs and the motion of simple bodies down an incline.

The upfront effort required to build specific analytical models is however offset by the relatively low complexity and high performance of the software implementations. Thus, Lagrangian models are often employed as “special cases” in tandem with more generic modelling approaches.

Hybrid systems are sometimes also employed whereby a system of bodies is controlled using a combination of analytical and numerical methods, subject to the current state of the system. A typical example is the analytical application of an equation of motion for a projectile whilst in mid-air, that switches to a numerically-based model to simulate collision with its target.

A distinct limitation of analytically-based models is that their implementation is most naturally represented in hard-coded form. In a development environment, where a number of such models must be introduced and fine-tuned, a substantial number of coding-compiling-testing cycles may be required. This limitation may however be overcome via the use of embedded scripting engines such as [**8**], which may be used to abstract the hard-coded implementation of an analytical model from the core code of the application.

1.2.2 Numerically-Based Modelling

Numerically-based modelling, based on *Newtonian mechanics*, foregoes the complexity of deriving analytical solutions for specific applications in favour of the basic application of the

Newtonian equations of motion using numerical integration, coupled with real-time collision detection and response, and the application of other constraints, such as mechanical joints and actuators.

Idealised rigid bodies in physics simulations typically assume a non-negligible volume and uniform mass density. Hence, the corresponding differential equations of motion must take into account the linear and angular effects of the applied forces. The equations are solved numerically using one of a possible number of methods (§ 2.4) as the simulation progresses forward in time by discrete time deltas.

The instantaneous linear and angular acceleration are derived from the forces and torques applied on the bodies with respect to their masses, and moments of inertia around their centres of mass. These quantities are then numerically integrated into the instantaneous linear and angular velocities and eventually into the position and orientation of the bodies via a further numerical integration step (§ 2.3).

1.3 Aggregate Mass Systems and Soft Body Modelling

In CPU-limited environments, such as older gaming consoles and the more recent hand-held systems, the equations may be simplified by treating a body as a series of point masses connected with very stiff, weightless springs [**9**], termed *aggregate mass systems*. Point masses eliminate the need of calculating moments of inertia, torques and other angular properties of motion. Applying a force to one of the point masses in the system results in a distribution of force to the other point masses along the interconnecting springs, and hence resulting in an overall torque over the system of point masses, thus approximating the effect of an offset force applied on a rigid body. The visual geometry is then positioned and oriented in accordance with the overall configuration of the aggregate mass, ignoring the deformation resulting from the interconnecting springs.

By relaxing the spring coefficients, aggregate mass systems may also be used to simulate soft bodies [**10**], using the point masses as control points to deform the visual geometry of the body. More sophisticated approaches for deformable bodies involve the use of regular lattices of springs and point masses in a technique termed the *finite element method* [**11**]

1.4 Collision Detection

Collision detection is arguably one of the most complex aspects of rigid and soft body dynamics in terms of techniques and computational effort required. Thus, the subject has been extensively studied and a wide variety of data structures and algorithms have been devised to counter this complexity [**12**].

Collision queries on the body geometry are performed in real-time on every simulation time step. Such queries entail intersection tests between the simple or composite geometric primitives constituting the shapes of the bodies when placed in world coordinates. In many real-time applications, such as computer games, complex visual geometry is physically approximated by relatively simple geometrical primitives such as boxes, spheres, cylinders, capsules and arbitrary convex polyhedrons (Figure 2). Collision queries on convex geometry are simpler to compute than for concave geometry and hence geometry of the former type is preferred for performance reasons. However, concave geometry may be represented by compositing convex primitives together.

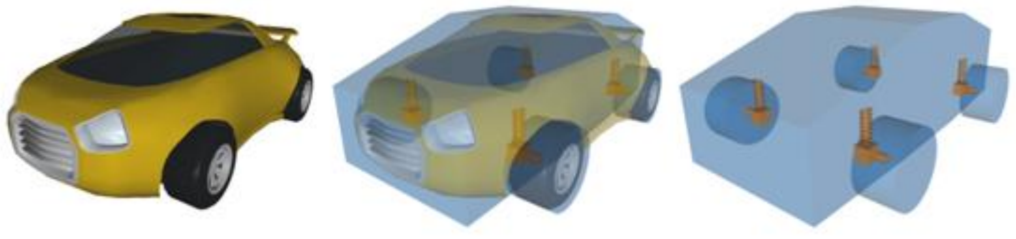


Figure 2 Approximating complex visual geometry by simple physical geometry

Constant time steps generally result in detecting collisions only after some interpenetration has occurred (Figure 3), in which case, the simulation may backed up to the point of contact using a time-bisecting algorithm [**13**]. Alternatively, the body positions may be fixed by linear projection to eliminate interpenetration (Figure 41). For more precise applications, continuous collision detection techniques [**14**], [**15**] are applied, whereby collision queries are also required to provide an estimated time of impact (TOI) in order to prevent the interpenetration of bodies or the *tunnelling* phenomenon in which relatively thin objects in simulations stepped in relatively large time deltas result in bodies passing through each other without collisions being detected. The TOI may hence be used to prioritise impending collision estimates and step the simulation to coincide with the estimates accordingly [**16**].

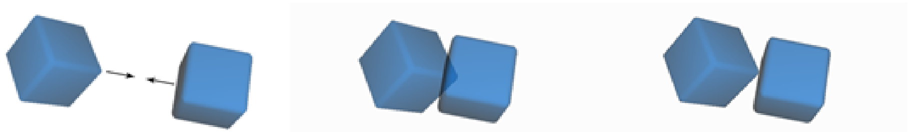


Figure 3 From left to right: on collision course; interpenetration; exact contact

The algorithmic complexity of collision detection often requires the use of spatial representation schemes to prune the collision search space sufficiently for real-time applications. This process is sometimes termed as *broad-phase collision detection*, as opposed to the *narrow-phase* pass characterised by exact body geometry queries. Examples of such representations include regular grids, grid hierarchies, octrees (Figure 4), KD-trees, coordinate sorting [**17**], binary space partition (BSP) trees [**18**], and bounding volume hierarchies (BVH’s) [**19**].

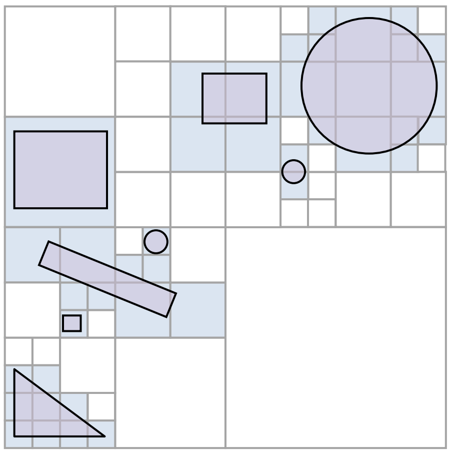


Figure 4 A 2D representation of an octree

The simplest scheme for culling the search space entails enclosing the bodies in simple bounding volumes such as spheres and axis-aligned boxes that act as geometric approximations. The collision detection sub-system thus tests the bounding volumes for intersection, in which case, the more complex body geometry may be tested (Figure 5). This scheme is typically used in tandem with broad-phase and narrow-phase collision detection, and is performed between these two phases in what is termed as *mid-phase collision detection*. Moreover, the broad-phase space partitioning schemes tend to manipulate the contained bodies in terms of their bounding volumes to facilitate the construction and periodic update of their data representations.

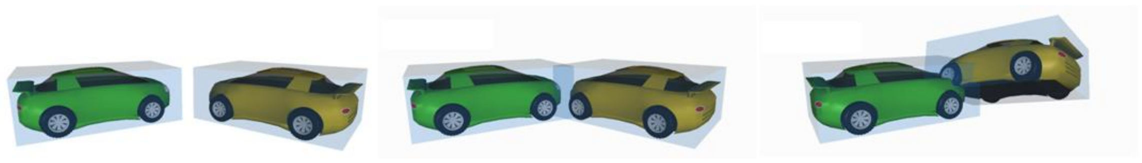


Figure 5 Bounding volumes, from left to right: no collision; potential collision; definite collision

1.5 Collision Response

Collision response is performed in real-time, whereby the changes to the velocities of the affected bodies are applied using the *energy conservation principle* (§ 2.6.1.1) to simulate the rebound. Collisions are generally resolved either by applying *instantaneous impulses* (§ 2.6.2) or through the application of temporary spring-like penalty forces (§ 2.6.3) at the points of contact. The former approach results in an instantaneous change in velocities, as would be expected from idealised rigid bodies. The latter approach results in a gradual change in velocities whereby the bodies may interpenetrate slightly for a number of simulation steps until the springs eventually push the bodies apart.

When bodies come into contact, they may intersect at one or more points, lines or areas on their respective surfaces (Figure 6), termed the contact manifold. This manifold is generally reduced to an equivalent set consisting only of contact points to simplify collision response (§ 2.5.3.1).

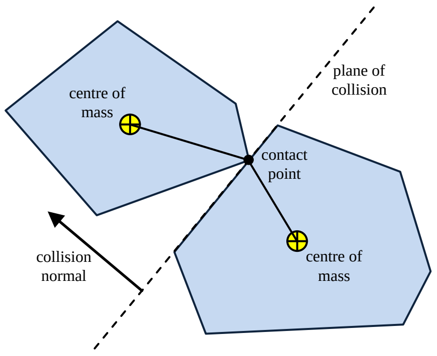


Figure 6 Parameters of interest in a simulated collision between two bodies

Simple configurations, such as two bodies colliding in space, may be resolved by applying collision resolution on the point of greatest penetration, based on assumption that this inherently solves contact at other less severe contact points. This approach however breaks down in more complex configurations such as bodies stacked on an immovable ground plane (Figure 7) under the force of gravity. In this case, the simulation is expected to maintain the system of bodies in static equilibrium. Naïve collision resolution in this case, results in a train of small scale collision responses up and down the stack as the bodies attempt to accelerate towards the ground. This often results in the bodies vibrating and eventually sliding off the stack, in what is commonly referred to as the *stacking problem* in the science of physics engine development.

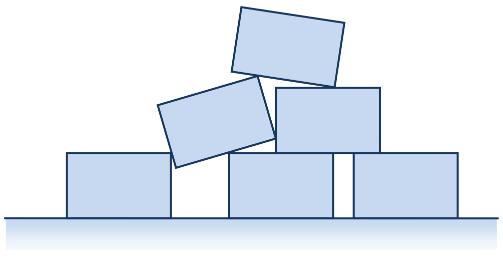


Figure 7 A typical configuration of bodies stacked on a ground plane

A related problem occurs when attempting to simulate a body in resting contact on an inclined plane where the motion is constrained by the interaction between the forces of gravity, reaction and static friction. An improper treatment of resting contacts in such a situation results in a stream of microscopic collision detection-response cycles that cause the body to bounce imperceptibly and eventually slide down the plane. Suitable treatment of such configurations are provided in research by [**20**] and [**13**], amongst others. These approaches seek solutions where all non-penetration constraints are satisfied simultaneously.

1.6 Joints and Actuators

In real life, articulated bodies may range from a simple mechanism consisting of a few moving parts, such as a pair of pliers, to a complex piece of machinery or a vertebrate organism composed of a mix of hard and soft materials elaborately jointed together. The joints in such bodies may not be clearly delineated, particularly in organic bodies: a human elbow consists of bones held in place by ligaments and with the whole assembly covered in muscles and other layers of flesh.

Articulated bodies in the context of physical simulation usually imply a collection of rigid bodies whose relative movement is limited by special constraints, commonly termed *joints*. Simulated soft bodies with limb-like appendages may arguably also be considered as articulated, however the nature of articulation in such bodies is considered a side effect of their gelatinous nature and hence are excluded from this category.

The techniques applied for collision response are often extended to implement articulated bodies. A joint or actuator between two bodies is essentially a constraint, or a number of constraints, imposed on the relative position and orientation (Figure 8) of the bodies .

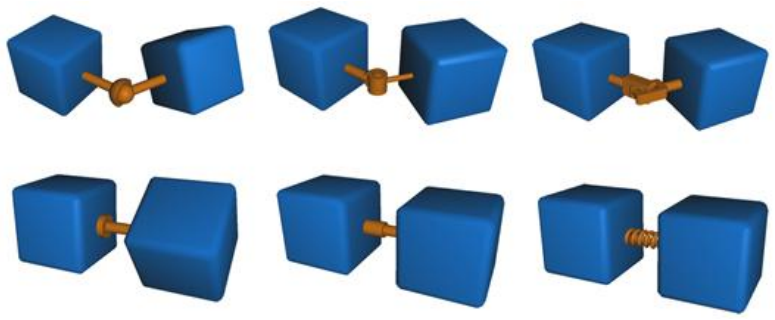


Figure 8 Common joint types, clockwise from top left: ball-and-socket; hinge; CV joint; spring; slider; axle

Joints are typically implemented by applying corrective forces or impulses to compensate for invalidated joint constraints on every simulation step [**21**] For example, a ball and socket joint may be simulated by imposing a minimum distance between specific points on two bodies. Such a constraint removes three out of the six relative degrees of freedom, allowing the objects to twist or bend in any direction relative to each other. More restrictive joints, such as continuous-velocity (CV) joints and hinges are implemented in a similar manner by further reducing the degrees of freedom. Many physics engine implementations, such as ODE [**22**], unify the concepts of non-penetration and joints under the general notion of constraints. Collision responses are handled by applying instantaneous non-penetration constraints at the points of contact.

The concept of jointed rigid bodies has obvious applications in the simulation of mechanical devices, such as cars or rope bridges consisting of rigid components connected with specific types of joints. For instance, the body of a car may be approximated by a cuboid connected to cylinders acting as wheels via axle joints or a swivelling wheel assembly (Figure 2). Similarly a bridge may be simulated by a number of plank-shaped cuboids connected with ball-and-socket or cable joints. On the other hand, an articulated organic body, such as a humanoid form, may not be simulated directly in this manner because of its seamless joints. Such a body may be however simulated via a skeletal animation technique termed *skinning*, whereby a visual three-dimensional (3D) model of the body is smoothly deformed according to an underlying simplified skeleton (Figure 9) consisting of jointed, rod-like rigid bodies.

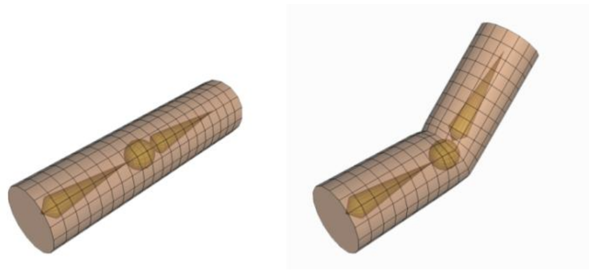


Figure 9 Simulating organic joints using skinning

1.7 Scaling Considerations

The overall algorithmic complexity of body dynamics simulation, and in particular collision detection, imposes limits on the complexity of simulations running in real-time. Spatial representation schemes alleviate these limitations; however, systems of bodies in equilibrium or near-equilibrium featuring many resting contacts, tend to generate a continuous stream of mini collisions [**20**] that, aside from potentially introducing instability, unnecessarily consume substantial processing resources.

One approach to mitigate this problem is to use a sleep-wake algorithm that identifies islands of bodies in contact that are static with respect to each other and disables collision detection and response for each island until it comes into contact with an external body, or until one of the bodies within the island is induced into motion by an explicit force (Figure 10). The method of identifying such islands usually entails keeping track of bodies in contact that are relatively static over a short period of time. This approach is used by a number of commercial and community-developed physics engine implementations such as SPE [**23**] and Bullet [**24**].

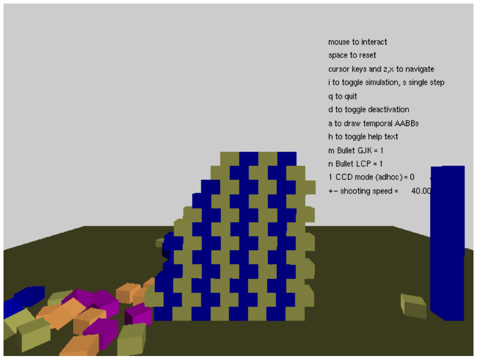

```
mouse to interact
space to reset
cursor keys and z,x to navigate
i to toggle simulation, s single step
q to quit
d to toggle deactivation
a to draw temporal AABBs
h to toggle help text
m Bullet GJK = 1
n Bullet LCP = 1
1 CCD mode (adhoc) = 0
+- shooting speed =     40.00
```

Figure 10 The sleep/wake algorithm in Bullet demonstrated using colour-coded bodies

Chapter 2
Background

Following our review on the techniques employed for physics simulations, in this chapter we outline the theoretical foundation and the mathematical constructs required to implement physics engines, with a focus on the specific requirements for the Gravitas engine. This chapter also provides a backdrop against which we justify the framework design decisions taken in Chapter 3, by considering a spectrum of existing simulation techniques with the purpose of identifying generalisations and areas of commonality upon which to base the physics engine framework.

2.1 Physics Engines

Technically, a physics engine is an interactive computer simulation of an idealised configuration of bodies in space whose state, at any given time, consists of the position and orientation, or collectively, the placement of bodies, and the forces acting upon them. These state variables are typically represented using 3D vectors, matrices and other such constructs. The rules that govern such a simulation are rooted in classical mechanics from which mathematical models are derived. However, the chaotic nature of an interactive simulation encourages the use of numeric methods, rather than traditional analytic methods, for solving these models.

2.2 Spatial Transformations

A physics engine typically operates in multiple coordinate systems, including the *absolute*, or *world*, frame of reference, the frames of reference local to each body and the collision frame of reference defined by the plane and normal of collision whenever one occurs. Thus, a crucial mathematical requirement of physics engines is the ability to switch from one coordinate frame to another. This requirement is satisfied by *affine* transformations of the form $\tau: \mathbb{R}^3 \longmapsto \mathbb{R}^3$ which may be defined in a number of ways as detailed in the subsequent sections.

2.2.1 Homogenous Matrix Transforms

Within the realm of 3D computer graphics, the de facto construct for describing a change in the position and orientation of objects is the affine transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times4}$ [**25**] of the form

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ is constrained to be an orthogonal rotation matrix, $\boldsymbol{t} \in \mathbb{R}^3$ is a column vector representing a linear translation. A point $\boldsymbol{p} \in \mathbb{R}^3$ may be transformed by $\boldsymbol{M}$ to yield $\boldsymbol{p}' \in \mathbb{R}^3$ using matrix-to-vector multiplication as follows

$$\begin{bmatrix}\boldsymbol{p}'\\1\end{bmatrix} = \boldsymbol{M}\begin{bmatrix}\boldsymbol{p}\\1\end{bmatrix} \tag{2.2}$$

where $\begin{bmatrix}\boldsymbol{p}\\1\end{bmatrix} \in \mathbb{R}^4$ are *homogenous coordinates* [**26**] consisting of the position vector of point $\boldsymbol{p}$ augmented with an additional unit coefficient to allow pre-multiplication by matrix $\boldsymbol{M}$. The transformed point $\boldsymbol{p}'$ is extracted from the resulting vector $\begin{bmatrix}\boldsymbol{p}'\\1\end{bmatrix} \in \mathbb{R}^4$. The transformation $\boldsymbol{M}$ effectively rotates the point $\boldsymbol{p}$ around the origin using the rotation sub-matrix $\boldsymbol{R}$ and translates the resulting point by an offset $\boldsymbol{t}$ to the new point $\boldsymbol{p}'$, that is

$$\boldsymbol{p}' = \boldsymbol{R}\boldsymbol{p} + \boldsymbol{t} \tag{2.3}$$

where points $\boldsymbol{p}$ and $\boldsymbol{p}'$ are treated as position vectors for the purpose of matrix multiplication. The principal motivation for the combined transformation matrix $\boldsymbol{M}$ stems from the convenient property that two such transformations may be combined by simply performing a matrix multiplication. Given two 4x4 matrix transforms $\boldsymbol{M}_1 = \begin{bmatrix}\boldsymbol{R}_1 & \boldsymbol{t}_1\\ \boldsymbol{0} & 1\end{bmatrix}$ and $\boldsymbol{M}_2 = \begin{bmatrix}\boldsymbol{R}_2 & \boldsymbol{t}_2\\ \boldsymbol{0} & 1\end{bmatrix}$, the composite transformation is given by

$$\begin{bmatrix}\boldsymbol{p}'\\1\end{bmatrix} = \boldsymbol{M}_2\boldsymbol{M}_1\begin{bmatrix}\boldsymbol{p}\\1\end{bmatrix}$$

which is equivalent to

$$\boldsymbol{p}' = \boldsymbol{R}_2(\boldsymbol{R}_1\boldsymbol{p} + \boldsymbol{t}_1) + \boldsymbol{t}_2$$

The homogenous 4x4 transformation matrix (2.1) may also be employed within physics engines to describe the position and orientation of bodies in space. This construct is however parameterised by sixteen variables which must be constrained down to the six degrees of freedom (three for position and three for orientation) required to describe the placement of a body in space. This representation thus has substantial overhead in terms of memory and computational requirements. This representation also suffers from numerical errors creeping in a running simulation due to the limited precision of digital computation, resulting for instance, in the rotation component $\boldsymbol{R}$ losing its orthonormality. The rotation sub-matrix is typically corrected using the Gram-Schmidt *orthonormalization process* [**27**]. The process computes three mutually orthogonal unit vectors $\hat{\boldsymbol{u}}, \hat{\boldsymbol{v}}, \hat{\boldsymbol{w}} \in \mathbb{R}^3$ similar to the original column vectors $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in (\mathbb{R}^3)^T$ of

$$\boldsymbol{R} = [\boldsymbol{u} \quad \boldsymbol{v} \quad \boldsymbol{w}] = \begin{bmatrix}u_x & v_x & w_x\\ u_y & v_y & w_y\\ u_z & v_z & w_z\end{bmatrix}$$

using the following equations

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{u}}{|\boldsymbol{u}|} \tag{2.4a}$$

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v} - \boldsymbol{v} \cdot \boldsymbol{u}'}{|\boldsymbol{v} - \boldsymbol{v} \cdot \boldsymbol{u}'|} \tag{2.4b}$$

$$\hat{\boldsymbol{w}} = \boldsymbol{u}' \times \boldsymbol{v}' \tag{2.4c}$$

$$\boldsymbol{R}' = [\hat{\boldsymbol{u}} \quad \hat{\boldsymbol{v}} \quad \hat{\boldsymbol{w}}] = \begin{bmatrix} u_x' & v_x' & w_x' \\ u_y' & v_y' & w_y' \\ u_z' & v_z' & w_z' \end{bmatrix} \tag{2.4d}$$

where $\boldsymbol{R}' \in \mathbb{R}^{3\times3}$ is the fixed orthonormal matrix. Informally, this orthonormalization procedure first scales back $\boldsymbol{u}$ to yield unit vector $\hat{\boldsymbol{u}}$. Next, it removes any residual component of $\boldsymbol{v}$ along $\hat{\boldsymbol{u}}$ and scales it back to yield unit vector $\hat{\boldsymbol{v}}$ orthogonal with $\hat{\boldsymbol{u}}$. Finally, unit vector $\hat{\boldsymbol{w}}$ is recalculated as the vector product of $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{v}}$, resulting in a unit-length vector orthogonal to both vectors.

2.2.2 Quaternion-Based Transforms

A more compact representation for an affine transform is the rotation matrix and translation vector pair $\tau = (\boldsymbol{R}, \boldsymbol{t})$ where $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ and $\boldsymbol{t} \in \mathbb{R}^3$, with the transform defined by

$$\tau(\boldsymbol{p}) \stackrel{\text{def}}{=} \boldsymbol{R}\boldsymbol{p} + \boldsymbol{t} \tag{2.5}$$

For a point in space $\boldsymbol{p} \in \mathbb{R}^3$ treated as a vector offset from the origin. This non-homogenous transform is parameterised by twelve variables and hence requires less memory and computational resources. For this representation, it is possible to construct an algebra such that composition of two such transforms still yields a transform consisting of an overall rotation followed by an overall translation. Given $\tau_1 = (\boldsymbol{R}_1, \boldsymbol{t}_1)$ and $\tau_2 = (\boldsymbol{R}_2, \boldsymbol{t}_2)$, the composition $\tau_2 \circ \tau_1$ may be defined as

$$\tau_2 \circ \tau_1(\boldsymbol{p}) \stackrel{\text{def}}{=} \boldsymbol{R}_2(\boldsymbol{R}_1\boldsymbol{p} + \boldsymbol{t}_1) + \boldsymbol{t}_2 = (\boldsymbol{R}_2\boldsymbol{R}_1)\boldsymbol{p} + (\boldsymbol{R}_2\boldsymbol{t}_1 + \boldsymbol{t}_2)$$

By isolating terms the composite transform may be defined in terms of a composite rotation and translation

$$\tau_2 \circ \tau_1 = (\boldsymbol{R}, \boldsymbol{t}) = (\boldsymbol{R}_2\boldsymbol{R}_1, \boldsymbol{R}_2\boldsymbol{t}_1 + \boldsymbol{t}_2) \tag{2.6}$$

Similarly, an inverse transform may be defined by writing out the equation for transforming $\boldsymbol{p}$ to $\boldsymbol{p}' \in \mathbb{R}^3$ and solving for $\boldsymbol{p}$

$$\tau(\boldsymbol{p}) = \boldsymbol{p}' = \boldsymbol{R}\boldsymbol{p} + \boldsymbol{t}$$

$$\begin{aligned} \boldsymbol{Rp} &= \boldsymbol{p}' - \boldsymbol{t} \\ \boldsymbol{p} &= \boldsymbol{R}^{-1}(\boldsymbol{p}' - \boldsymbol{t}) \\ &= \boldsymbol{R}^{-1}\boldsymbol{p}' + (-\boldsymbol{R}^{-1}\boldsymbol{t}) \end{aligned}$$

By isolating terms, the inverse is defined as

$$\tau^{-1} = (\boldsymbol{R}^{-1}, -\boldsymbol{R}^{-1}\boldsymbol{t}) \tag{2.7}$$

The rotation component $\boldsymbol{R}$ in the above representation still suffers from redundant parameters and numerical stability issues. A more compact representation for the rotation component is the *quaternion* (Appendix A), first introduced as a rotational transform in computer animation by [**28**]. A quaternion $\boldsymbol{q} \in \mathbb{H}$ is parameterised by four variables, which must be constrained to the three degrees of freedom representing orientation. This is achieved by constraining the quaternion to unit norm through the equation $|\boldsymbol{q}| = 1$ (A.6). Geometrically, the set of all possible rotation quaternions thus consists of all points on a 4D hypersphere (Figure 11) of unit radius.

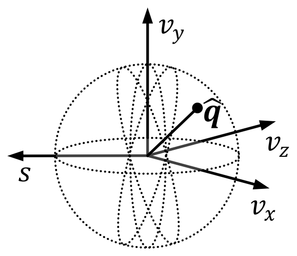


Figure 11 Unit quaternion represented as a point on the unit 4D hypersphere

A quaternion in the form $\hat{\boldsymbol{q}} = \left(\cos\frac{\theta}{2}, \hat{\boldsymbol{n}}\sin\frac{\theta}{2}\right)$ where the $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ is a unit vector and $\theta \in \mathbb{R}$, represents a *right-handed* rotation around an axis in the direction $\hat{\boldsymbol{n}}$ by the angle θ in radians. It should be noted that this form satisfies the unit norm constraint because

$$|\hat{\boldsymbol{q}}| = \left|\left(\cos\frac{\theta}{2}, \hat{\boldsymbol{n}}\sin\frac{\theta}{2}\right)\right| = \sqrt{\cos^2\frac{\theta}{2} + \hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{n}}\sin^2\frac{\theta}{2}} = \sqrt{\cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}} = 1$$

A vector $\boldsymbol{v} \in \mathbb{R}^3$ may be rotated using the rotation quaternion $\hat{\boldsymbol{q}}$ to yield a new vector $\boldsymbol{v}' \in \mathbb{R}^3$ using quaternion multiplication (A.5b) as follows

$$(s', \boldsymbol{v}') = \hat{\boldsymbol{q}}(0, \boldsymbol{v})\hat{\boldsymbol{q}}^* \tag{2.8}$$

In the equation for rotation, $\boldsymbol{v}$ is “wrapped” into a quaternion $(0, \boldsymbol{v}) \in \mathbb{H}$ to allow pre-multiplication by $\hat{\boldsymbol{q}}$ and post-multiplication by its conjugate $\hat{\boldsymbol{q}}^*$.

The rotated vector $\boldsymbol{v}'$ is extracted from the imaginary components of the resulting quaternion $(s', \boldsymbol{v}') \in \mathbb{H}$ while the real component $s' \in \mathbb{R}$ is discarded. For convenience in the subsequent formulations, the rotation equation $(s', \boldsymbol{v}') = \hat{\boldsymbol{q}}(0, \boldsymbol{v})\hat{\boldsymbol{q}}^*$ is written in the simplified form

$$\boldsymbol{v}' = \hat{\boldsymbol{q}}\boldsymbol{v}\hat{\boldsymbol{q}}^* \tag{2.9}$$

where the quaternion wrapping and unwrapping are implicitly assumed such that the unit quaternion $\hat{\boldsymbol{q}}$ acts as a rotation operator applied to a vector $\boldsymbol{v}$ to yield a new vector $\boldsymbol{v}'$. Similarly, a point $\boldsymbol{p} \in \mathbb{R}^3$ may be treated as a position vector and may thus be rotated around the origin to yield point $\boldsymbol{p}' \in \mathbb{R}^3$ using

$$\boldsymbol{p}' = \hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^* \tag{2.10}$$

Given (2.10), the rotation matrix $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ of the composite transform (2.5) may be replaced with the rotation quaternion $\hat{\boldsymbol{q}} \in \mathbb{H}$, yielding a more compact form of the transform $\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t}) \in \mathbb{H} \times \mathbb{R}^3$. This further reduces the number of parameters from twelve to seven, which is only one extra parameter beyond the cardinal six degrees of freedom. Assuming the simplified quaternion rotation operation (2.10), the transform applied to a point $\boldsymbol{p}$ is defined as

$$\tau(\boldsymbol{p}) \stackrel{\text{def}}{=} \hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^* + \boldsymbol{t} \tag{2.11}$$

Quaternions still suffer from numerical stability issues, whereby the norm of the rotation component diverges away from unity over time. However this may be mitigated by renormalizing the quaternion using

$$\hat{\boldsymbol{q}} = \frac{1}{|\boldsymbol{q}|}\boldsymbol{q} \tag{2.12}$$

The renormalisation computations required in (2.12) are obviously simpler than (2.4a), (2.4b) and (2.4c) applied to rotation matrices, and thus provide a further justification for the use of quaternions as rotational transforms.

We propose an algebra for $\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t})$ equivalent to that derived for $\tau = (\boldsymbol{R}, \boldsymbol{t})$ using similar arguments, but replacing matrix algebra with quaternion algebra. The composition of two transforms $\tau_1 = (\hat{\boldsymbol{q}}_1, \boldsymbol{t}_1)$ and $\tau_2 = (\hat{\boldsymbol{q}}_2, \boldsymbol{t}_2)$ is given by

$$\tau_2 \circ \tau_1 = \tau = (\hat{\boldsymbol{q}}, \boldsymbol{t}) = (\hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1, \hat{\boldsymbol{q}}_2\boldsymbol{t}_1\hat{\boldsymbol{q}}_2^* + \boldsymbol{t}_2) \tag{2.13}$$

The derivation follows by applying transform τ_1 to a point $\boldsymbol{p}$ followed by transform τ_2 and collecting and isolating terms using quaternion operations (A.1), (A.2), (A.5b) and (A.10), as follows

$$\begin{aligned} \tau_2 \circ \tau_1(\boldsymbol{p}) &= \hat{\boldsymbol{q}}_2(\hat{\boldsymbol{q}}_1\boldsymbol{p}\hat{\boldsymbol{q}}_1^* + \boldsymbol{t}_1)\hat{\boldsymbol{q}}_2^* + \boldsymbol{t}_2 \\ &= \hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1\boldsymbol{p}\hat{\boldsymbol{q}}_1^*\hat{\boldsymbol{q}}_2^* + \hat{\boldsymbol{q}}_2\boldsymbol{t}_1\hat{\boldsymbol{q}}_2^* + \boldsymbol{t}_2 \end{aligned}$$

$$= (\hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1)\boldsymbol{p}(\hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1)^* + (\hat{\boldsymbol{q}}_2\boldsymbol{t}_1\hat{\boldsymbol{q}}_2^* + \boldsymbol{t}_2)$$
$$= \hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^* + \boldsymbol{t}$$

It should be noted the set of unit quaternions is closed under quaternion multiplication, which follows from $|\hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1| = |\hat{\boldsymbol{q}}_2||\hat{\boldsymbol{q}}_1| = 1$ by (A.7). Thus, the quaternion product $\hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1$ is itself a unit quaternion and hence, it is possible to isolate the rotation quaternion term $\hat{\boldsymbol{q}} = \hat{\boldsymbol{q}}_2\hat{\boldsymbol{q}}_1$ of the composite transform.

The term $\hat{\boldsymbol{q}}_2\boldsymbol{t}_1\hat{\boldsymbol{q}}_2^* + \boldsymbol{t}_2$, according to the simplified rotation operator (2.10), evaluates to a translation vector $\boldsymbol{t} \in \mathbb{R}^3$ of the composite transform.

The transform inverse τ^{-1} of a transform $\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t})$ is given by

$$\tau^{-1} = (\hat{\boldsymbol{q}}^*, -\hat{\boldsymbol{q}}^*\boldsymbol{t}\hat{\boldsymbol{q}}) \tag{2.14}$$

The derivation follows by applying the transform τ to a point $\boldsymbol{p}$ to yield the new point $\boldsymbol{p}'$, solving for $\boldsymbol{p}$ and isolating terms, as follows

$$\begin{aligned}
\boldsymbol{p}' &= \hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^* + \boldsymbol{t} \\
\hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^* &= \boldsymbol{p}' - \boldsymbol{t} \\
\hat{\boldsymbol{q}}^*\hat{\boldsymbol{q}}\boldsymbol{p}\hat{\boldsymbol{q}}^*\hat{\boldsymbol{q}} &= \hat{\boldsymbol{q}}^*(\boldsymbol{p}' - \boldsymbol{t})\hat{\boldsymbol{q}} \\
\boldsymbol{p} &= \hat{\boldsymbol{q}}^*\boldsymbol{p}'\hat{\boldsymbol{q}} - \hat{\boldsymbol{q}}^*\boldsymbol{t}\hat{\boldsymbol{q}} \\
&= \hat{\boldsymbol{q}}^*\boldsymbol{p}'(\hat{\boldsymbol{q}}^*)^* + (-\hat{\boldsymbol{q}}^*\boldsymbol{t}\hat{\boldsymbol{q}})
\end{aligned}$$

This result makes use of (A.9) for $|\boldsymbol{q}| = 1$ and hence $\hat{\boldsymbol{q}}\hat{\boldsymbol{q}}^* = 1$.

The above derivations complete the algebra for compact quaternion-based isomorphic transformations of the form $\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t})$.

A useful result is the conversion of a rotation quaternion $\hat{\boldsymbol{q}} = [s \quad v_x \quad v_y \quad v_z]$ to a rotation matrix $\boldsymbol{R}$, given by

$$\boldsymbol{R} = \begin{bmatrix} 1 - 2{v_y}^2 - 2{v_z}^2 & 2v_xv_y - 2v_zs & 2v_xv_z + 2v_ys \\ 2v_xv_y + 2v_zs & 1 - 2{v_x}^2 - 2{v_z}^2 & 2v_yv_z - 2v_xs \\ 2v_xv_z - 2v_ys & 2v_yv_z + 2v_xs & 1 - 2{v_x}^2 - 2{v_y}^2 \end{bmatrix} \tag{2.15}$$

Hence, the conversion from a transform

$$\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t}) = ([s \quad v_x \quad v_y \quad v_z], [t_x \quad t_y \quad t_z])$$

to a homogenous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times4}$ is given by

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} = \begin{bmatrix} 1 - 2{v_y}^2 - 2{v_z}^2 & 2v_x v_y - 2v_z s & 2v_x v_z + 2v_y s & t_x \\ 2v_x v_y + 2v_z s & 1 - 2{v_x}^2 - 2{v_z}^2 & 2v_y v_z - 2v_x s & t_y \\ 2v_x v_z - 2v_y s & 2v_y v_z + 2v_x s & 1 - 2{v_x}^2 - 2{v_y}^2 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

Conversely, the construction a quaternion-based transform τ, from the homogenous transformation matrix $\boldsymbol{M} \in \mathbb{R}^{4\times4}$ defined as in (2.1), is given by

$$\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t}) = \big((s, \boldsymbol{v}), [t_x \quad t_y \quad t_z]\big) \tag{2.16a}$$

where

$$s = \frac{1}{2}\sqrt{1 + r_{11} + r_{22} + r_{33}} \tag{2.16b}$$

and

$$\boldsymbol{v} = \frac{1}{4s}[r_{32} - r_{23} \quad r_{13} - r_{31} \quad r_{21} - r_{12}] \tag{2.16c}$$

The derivations (2.15) and (2.16a) to (2.16c) find practical application when rendering the simulation on a graphics device, which typically requires the use of homogenous matrix transforms (2.1) to place visual models in the world frame of reference.

2.3 Kinetics

The term *kinetics* refers to a branch of dynamics that concerns itself with the motion of bodies under the effect of forces. In the context of the physics engine, kinetics deals with the effect of the linear and angular acceleration, resulting from the applied forces, on a body's velocity and position over time.

Equations for motion are usually expressed as ordinary differential equations. In terms of the physics engine implementation, a *numerical solver* is employed whereby the acceleration for the current moment in time is computed from the acting forces, integrated to yield the current velocity, and integrated once more to yield the current position.

2.3.1 Linear Motion

The laws of linear motion govern the effect of translational forces acting on a body, and resulting in motion that does not include any rotational components.

2.3.1.1 Forces and Mass

For an idealised, dimensionless point mass $m \in \mathbb{R}$ with forces $\boldsymbol{f}_i \in \mathbb{R}^3$ acting upon it, such that $1 \le i \le n$ and $i, n \in \mathbb{N}$, the acceleration $\boldsymbol{a} \in \mathbb{R}^3$ of the point mass is given by

$$\boldsymbol{a} = \frac{1}{m}\sum_{i=1}^{n}\boldsymbol{f}_i = \frac{\boldsymbol{f}}{m} \tag{2.17}$$

The equation is an aspect of *d'Alembert's principle* [**29**], stating that all the forces acting on the point mass can be summed up into a single force $\boldsymbol{f} \in \mathbb{R}^3$ acting on the mass, which multiplied by the mass inverse yields its acceleration. Equation (2.17) further indicates that for a given force a body with more mass will accelerate more slowly than a body with less mass, in line with the intuitive understanding of kinetic inertia.

The linear force equation (2.17) also applies to point mass aggregates and solid masses, provided that their overall mass is known and that the forces are applied such that they do not induce rotational acceleration.

2.3.1.2 Linear Kinetic Equations of Motion

The velocity $\boldsymbol{v} \in \mathbb{R}^3$, parameterised by time $t \in \mathbb{R}$, is related to the acceleration $\boldsymbol{a}$ by the differential equation

$$\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt} = \dot{\boldsymbol{v}} \tag{2.18a}$$

Hence, given an initial known velocity $\boldsymbol{v}_0 \in \mathbb{R}^3$ of a body, its velocity for the current time t may be computed analytically as

$$\boldsymbol{v} = \boldsymbol{v}_0 + \int_0^t \boldsymbol{a}dt \tag{2.18b}$$

Similar equations relate the body's velocity with its position vector $\boldsymbol{x} \in \mathbb{R}^3$ in space

$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt} = \dot{\boldsymbol{x}} \tag{2.19a}$$

$$\boldsymbol{x} = \boldsymbol{x}_0 + \int_0^t \boldsymbol{v}dt \tag{2.19b}$$

where $\boldsymbol{x}_0 \in \mathbb{R}^3$ is the position vector of the body at time $t = 0$.

2.3.2 Angular Motion

The above equations are sufficient for describing the motion of independent point masses, as the notion of rotation does not apply. For discrete point mass aggregates consisting of point masses rigidly connected together, or bodies with a continuously distributed mass spread over a volume in space, the equations of motion must be extended to cover rotation. Besides linear acceleration, forces acting on solid bodies induce rotation around a particular point in the body, known as the *centre of mass* (CM). Thus, the motion of a solid body may be described in terms of the linear motion of its CM and the angular motion of the body around the centroid.

2.3.2.1 Centre of Mass

The centre of mass is a point, usually within the body itself, such that any plane passing through the point bisects the body in two equal masses. For a discrete aggregate of n point masses, defined by $(m_i, \boldsymbol{x}_i)$ for $1 \leq i \leq n$ and $i, n \in \mathbb{N}$, where $m_i \in \mathbb{R}$ and $\boldsymbol{x}_i \in \mathbb{R}^3$ are, respectively, the mass and position of ith point mass, the CM $\boldsymbol{x}_c \in \mathbb{R}^3$ is computed as follows

$$\boldsymbol{x}_c = \frac{\sum_{i=1}^{n} m_i \boldsymbol{x}_i}{\sum_{i=1}^{n} m_i} \tag{2.20}$$

Informally, the CM is computed as an average of the individual mass positions, weighted by their corresponding masses. For a solid body with a continuous mass density function $\rho: \boldsymbol{\chi} \mapsto \mathbb{R}$ mapping a density value for every point $\boldsymbol{x} \in \boldsymbol{\chi} \subset \mathbb{R}^3$ where $\boldsymbol{\chi}$ is the region of space occupied by the body, the discrete summations are replaced by volume integrals

$$\boldsymbol{x}_c = \frac{\oiiint_{\chi} \rho(\boldsymbol{x})\boldsymbol{x}d\boldsymbol{x}}{\oiiint_{\chi} \rho(\boldsymbol{x})d\boldsymbol{x}} = \frac{1}{m}\oiiint_{\chi} \rho(\boldsymbol{x})\boldsymbol{x}d\boldsymbol{x} \tag{2.21}$$

where $m \in \mathbb{R}$ is the total mass of the body. For a body with uniform mass density, $\rho(\boldsymbol{x})$ is constant and hence the equation simplifies to

$$\boldsymbol{x}_c = \frac{\oiiint_{\chi} \boldsymbol{x}d\boldsymbol{x}}{\oiiint_{\chi} d\boldsymbol{x}} = \frac{1}{v}\oiiint_{\chi} \boldsymbol{x}d\boldsymbol{x} \tag{2.22}$$

where $v \in \mathbb{R}$ is effectively the volume of $\boldsymbol{\chi}$. In this case, the CM $\boldsymbol{x}_c$ coincides with the *geometric centroid* of the volume occupied by the body. For practical physics engine applications, constant mass density is often assumed and hence the centroid equation may be used. Computing the centroid for geometric primitives, such as spheres, cuboids and cylinders is a trivial process due to the symmetry of these shapes. For more *ad hoc* geometry, such as arbitrarily polyhedra, an exact iterative algorithm [**30**], [**31**] or an approximate point sampling algorithm may be employed.

The use of point sampling to compute approximations for the mass properties, such as overall mass and the centroid, of a geometric entity essentially entails sampling points at regular intervals defined by a grid (Figure 12) in the region containing the geometry. If the sampled point is contained in the geometry, it is approximated by a point mass whose value is proportional to the volume of an individual grid cell. The required mass property is then computed as an aggregate of the point masses that pass the containment test. The overall mass is computed as a discrete sum of the individual point mass approximations, while the centroid is computed using (2.20). A more accurate result may be attained by treating each qualifying sample as a cubic element and computing its moment of inertia contribution using the Parallel Axis Theorem represented by equation (2.33) and the moment of inertia computation (2.37b) applied to the cube.

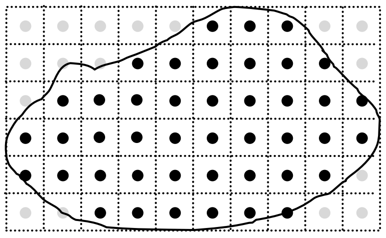


Figure 12 Regular point sampling for approximate mass property computation

2.3.2.2 Torque and Moment of Inertia

A force $\boldsymbol{f}$ applied to a solid body acts both as a translational force applied at the CM, and also as a rotational force $\boldsymbol{\tau} \in \mathbb{R}^3$, termed a *torque*, that is a function of the force vector and the offset vector $\boldsymbol{r} \in \mathbb{R}^3$ of the point of application $\boldsymbol{x}_f \in \mathbb{R}^3$ from the CM $\boldsymbol{x}_c$, computed as

$$\boldsymbol{\tau} = \boldsymbol{r} \times \boldsymbol{f} = (\boldsymbol{x}_f - \boldsymbol{x}_c) \times \boldsymbol{f} \tag{2.23}$$

The direction of the torque vector $\boldsymbol{\tau}$ represents the axis of the rotational force, while the magnitude represents the rotational intensity. It should be noted that when $\boldsymbol{r}$ is parallel to $\boldsymbol{f}$, $\boldsymbol{\tau} = \boldsymbol{0}$. This implies that a force acting anywhere along a line coincident with the CM does not induce torque. In addition, the vector product in (2.23) suggests that, keeping the point of application $\boldsymbol{x}_f$ fixed, the force $\boldsymbol{f}$ induces the greatest torque when $\boldsymbol{f}$ is perpendicular to $\boldsymbol{r}$.

Like its linear counterpart (2.17), the effect of a torque on a body is dependent on the angular equivalent of mass, termed the *moment of inertia*. The moment of inertia is a measure of the distribution of a body's mass with respect to some axis. For a single point mass $m \in \mathbb{R}$, the moment of inertia $I \in \mathbb{R}$ may be computed as a scalar using

$$I = mr^2 \tag{2.24}$$

where $r \in \mathbb{R}$ is the perpendicular distance from the axis in consideration.

In scalar terms, an applied torque $\tau \in \mathbb{R}$ is related to the angular acceleration $\alpha \in \mathbb{R}$ and moment of inertia I by

$$\alpha = \frac{\tau}{I} \tag{2.25}$$

where all three variables are taken in the context of the same axis.

For an n-point mass aggregate, the moment of inertia is computed as

$$I = \sum_{i=1}^{n} m_i {r_i}^2 = \sum_{i=1}^{n} I_i \tag{2.26}$$

where $m_i, r_i, I_i \in \mathbb{R}$ are, respectively, the mass, axis distance and individual moment of inertia of the ith point mass. For a body of continuous mass, the discrete summation is replaced by an integral involving the mass density function $\rho(\boldsymbol{x})$ over the region $\boldsymbol{\chi}$ occupied by the body

$$I = \oiiint_{\boldsymbol{\chi}} \rho(\boldsymbol{x})\, r^2 d\boldsymbol{x} \tag{2.27}$$

where $r \in \mathbb{R}$, dependent on $\boldsymbol{x} \in \mathbb{R}^3$, is the perpendicular distance of integral element $\boldsymbol{x}$ from the axis in consideration. An axis may be defined as an infinite line $\boldsymbol{s}(t) = \boldsymbol{p} + t\hat{\boldsymbol{d}}$, where $\boldsymbol{p} \in \mathbb{R}^3$ is a given point on this line, $\hat{\boldsymbol{d}} \in \mathbb{R}^3$ is a direction unit vector and $t \in \mathbb{R}$ parameterises the line for $-\infty \leq t \leq \infty$. The squared perpendicular distance r^2 can rewritten in terms of $\boldsymbol{x}$ as follows

$$r^2 = (\boldsymbol{x} - \boldsymbol{p})^2 - \left((\boldsymbol{x} - \boldsymbol{p}) \cdot \hat{\boldsymbol{d}}\right)^2 \tag{2.28}$$

The Parallel Axis Theorem relates the moment of inertia $I_c \in \mathbb{R}$ of a body with mass m, around an axis passing through the CM, with the inertia $I_d \in \mathbb{R}$ around some other axis parallel to it, separated by a distance r, by the equation

$$I_d = I_c + mr^2 \tag{2.29}$$

A corollary of this theorem, related to the implementation of the engine, is that for a given axis, the moment of inertia I_c may be pre-computed, whilst I_d may be computed in real-time using the above equation. Another application is for computing the moment of inertia for a composite body, in which case, r becomes an offset of a body component from some origin of the composite body.

The moment of inertia of a body in general varies depending on the axis for which it is computed. A convenient way to represent a unified moment of inertia value suitable for any axis is the *inertia tensor* matrix $\boldsymbol{I} \in \mathbb{R}^{3\times3}$, in the form

$$\boldsymbol{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}$$

where I_{ij} represents the moment of inertia around the jth axis when the body rotates around ith axis. The diagonal elements I_{ii} are effectively the moment of inertia values around the three principal axes, while the other elements are termed the *products of inertia*, which are usually zero for symmetric objects aligned with the principal axes and CM coincident with the origin, in which case, the tensor takes the simpler form

$$\boldsymbol{I} = \begin{bmatrix} I_{11} & 0 & 0 \\ 0 & I_{22} & 0 \\ 0 & 0 & I_{33} \end{bmatrix}$$

For an n-point mass aggregate, the elements $I_{ij} \in \mathbb{R}$ are defined as

$$I_{ij} = \sum_{k=1}^{n} m_k \left(\delta_{ij} \boldsymbol{r}_k{}^2 - r_{ki} r_{kj} \right) \tag{2.30}$$

where $1 \leq i, j \leq 3 \in \mathbb{N}$ index the principal axes, $m_k \in \mathbb{R}$ is the kth point mass, $\boldsymbol{r}_k = [r_{k1} \quad r_{k2} \quad r_{k3}] \in \mathbb{R}^3$ is the vector offset of point mass k from the point around which the tensor is calculated, and δ_{ij} is the Kronecker delta such that $\delta_{ii} = 1$ and $\delta_{ij} = 0$ for $i \neq j$.

For a continuous body, an integral form of the tensor computation applies

$$\boldsymbol{I} = \oiiint_{\chi} \rho(\boldsymbol{x})(\boldsymbol{r}^2 \boldsymbol{1} - \boldsymbol{r} \otimes \boldsymbol{r}) d\boldsymbol{x} \tag{2.31}$$

where $\boldsymbol{1} \in \mathbb{R}^{3\times3}$ is the 3x3 identity matrix, $\boldsymbol{r} \in \mathbb{R}^3$ is the vector offset of the element around which the tensor is calculated, and $\boldsymbol{r} \otimes \boldsymbol{r}$ is the outer cross product of $\boldsymbol{r}$ with itself, defined as

$$\boldsymbol{r} \otimes \boldsymbol{r} = \boldsymbol{r}^T \boldsymbol{r} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} [r_1 \quad r_2 \quad r_3] = \begin{bmatrix} r_1 r_1 & r_1 r_2 & r_1 r_3 \\ r_2 r_1 & r_2 r_2 & r_2 r_3 \\ r_3 r_1 & r_3 r_2 & r_3 r_3 \end{bmatrix}$$

For constant density ρ, the tensor equation simplifies to

$$\boldsymbol{I} = \rho \oiiint_{\chi} (\boldsymbol{r}^2 \boldsymbol{1} - \boldsymbol{r} \otimes \boldsymbol{r}) d\boldsymbol{x} = \rho \boldsymbol{D} \tag{2.32}$$

where $\boldsymbol{D} \in \mathbb{R}^{3\times3}$, which we term as a *distribution tensor*, represents an inertia tensor per unit mass density. This tensor finds application as a means of calculating the inertia tensor for a geometric construct when no density function is given and uniform density is assumed.

The Parallel Axis Theorem may also be unified in tensor form as follows

$$\boldsymbol{I}_d = \boldsymbol{I}_c + m(\boldsymbol{r}^2 \boldsymbol{1} - \boldsymbol{r} \otimes \boldsymbol{r}) \tag{2.33}$$

where $\boldsymbol{I}_d \in \mathbb{R}^{3\times3}$ is the inertia tensor displaced by the vector offset $\boldsymbol{r} \in \mathbb{R}^3$ from the CM, $\boldsymbol{I}_c \in \mathbb{R}^{3\times3}$ is the tensor computed around the CM, and $m \in \mathbb{R}$ is the overall mass of the body.

The inertia tensor conveniently unifies the equation relating the vector form of the angular acceleration $\boldsymbol{\alpha} \in \mathbb{R}^3$ and torque $\boldsymbol{\tau} \in \mathbb{R}^3$ as follows

$$\boldsymbol{\alpha} = \boldsymbol{I_w}^{-1}\boldsymbol{\tau} \tag{2.34}$$

where $\boldsymbol{I_w} \in \mathbb{R}^{3\times3}$ is the inertia tensor computed according to the body's orientation in space. In a real-time simulation, $\boldsymbol{I_w}$ must be repeatedly computed in line with the body's changing orientation. Performing this computation from scratch is usually expensive in terms of processing time. A solution to this problem, applicable to rigid bodies, pre-computes the tensor $\boldsymbol{I_b} \in \mathbb{R}^{3\times3}$ for the body geometry placed at the origin and aligned with the principal axes. During the simulation, $\boldsymbol{I_w}$ is computed by transforming the tensor $\boldsymbol{I_b}$ to $\boldsymbol{I_w}$ using the equation

$$\boldsymbol{I_w} = \boldsymbol{R}\boldsymbol{I_b}\boldsymbol{R}^{-1} \tag{2.35}$$

where $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ is the rotation transform computed from the body's orientation quaternion $\boldsymbol{q} \in \mathbb{H}$. At runtime, the computer simulation operates only with inverse tensor matrices, which are expensive to compute on every simulation step. Hence, to further speed up the computation for rigid bodies, the inverse $\boldsymbol{I_b}^{-1}$ of the inertia tensor in body coordinates $\boldsymbol{I_b}$ is pre-computed, and the inverse tensor in world coordinates is dynamically computed using

$$\boldsymbol{I_w}^{-1} = \boldsymbol{R}\boldsymbol{I_b}^{-1}\boldsymbol{R}^{-\mathbf{1}} \tag{2.36}$$

2.3.2.3 Common Inertia Tensor Results

As in the case of mass and centroid computation, deriving inertia tensors for simple geometric primitives aligned with the principal axes and with their centroid coinciding with the origin, is a relatively simple process. Due to symmetry, the tensors contain no products of inertia and the diagonal elements may be computed individually for each axis by identifying a suitable integration element. The following are some inertia tensor results for common geometric primitives, given in distribution tensor form.

A sphere (Figure 13) remains invariant under any rotation, hence, its moment of inertia around any arbitrary axis is constant and its products of inertia are zero.

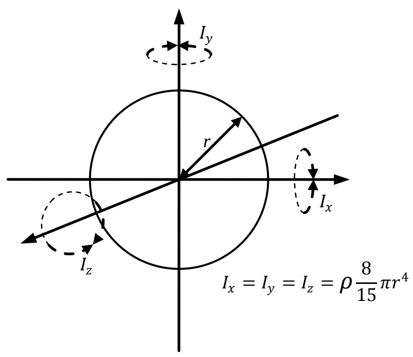
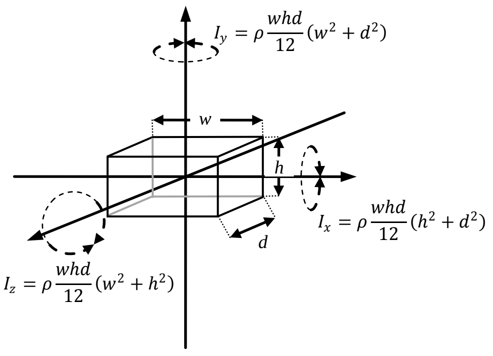


Figure 13 Moments of inertia for a sphere around the principal axes

Thus, the inertia tensor for a sphere of radius $r \in \mathbb{R}$, and uniform density $\rho \in \mathbb{R}$ is

$$\boldsymbol{I} = \rho\boldsymbol{D} = \rho\begin{bmatrix} \frac{8}{15}\pi r^4 & 0 & 0 \\ 0 & \frac{8}{15}\pi r^4 & 0 \\ 0 & 0 & \frac{8}{15}\pi r^4 \end{bmatrix} \tag{2.37a}$$

Due to symmetry, the inertia tensor of a cuboid (Figure 14), or box, aligned with the principal axes exhibits no products of inertia. However, the moment of inertia around each principal axis depends on the box dimensions.



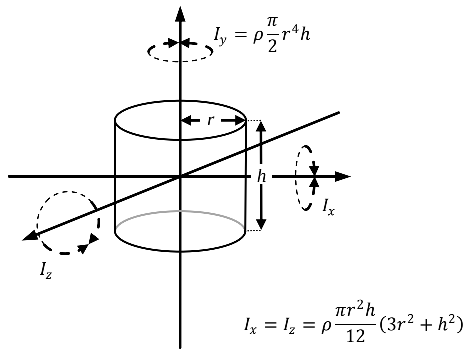
$I_y = \rho\frac{whd}{12}(w^2 + d^2)$
w
h
$I_x = \rho\frac{whd}{12}(h^2 + d^2)$
d
$I_z = \rho\frac{whd}{12}(w^2 + h^2)$

Figure 14 Moments of inertia for a box around the principal axes

The inertia tensor for a box with width, height and depth $w, h, d \in \mathbb{R}$, of uniform density $\rho \in \mathbb{R}$ is

$$\boldsymbol{I} = \rho\boldsymbol{D} = \rho\begin{bmatrix} \frac{whd}{12}(h^2 + d^2) & 0 & 0 \\ 0 & \frac{whd}{12}(w^2 + d^2) & 0 \\ 0 & 0 & \frac{whd}{12}(w^2 + h^2) \end{bmatrix} \tag{2.37b}$$

An axis-aligned cylinder (Figure 15) also exhibits symmetry around the principal axes, and hence the products of inertia are zero. The two principal moments of inertia are computed around the main axis running through the cylinder centre, and any axis running through the plane containing the centroid and orthogonal to the main axis.



$I_y = \rho\frac{\pi}{2}r^4h$
r
h
I_x
I_z
$I_x = I_z = \rho\frac{\pi r^2h}{12}(3r^2 + h^2)$

Figure 15 Moments of inertia for a cylinder around the principal axes

Hence, the inertia tensor for a cylinder aligned with the y-axis with radius $r \in \mathbb{R}$ and height $h \in \mathbb{R}$, of uniform density $\rho \in \mathbb{R}$ is

$$\boldsymbol{I} = \rho\boldsymbol{D} = \rho\begin{bmatrix} \frac{\pi r^2 h}{12}(3r^2 + h^2) & 0 & 0 \\ 0 & \frac{\pi}{2}r^4 h & 0 \\ 0 & 0 & \frac{\pi r^2 h}{12}(3r^2 + h^2) \end{bmatrix} \tag{2.37c}$$

The inertia tensor for a convex polyhedron of arbitrary complexity may be computed using an iterative algorithm as per [**30**] or by using the point sampling technique described in (§ 2.3.2.1).

2.3.2.4 Angular Kinetic Equations of Motion

The equation that relates angular acceleration $\boldsymbol{\alpha} \in \mathbb{R}^3$ to the angular velocity $\boldsymbol{\omega} \in \mathbb{R}^3$ of a body around its CM is

$$\boldsymbol{\alpha} = \frac{d\boldsymbol{\omega}}{dt} = \dot{\boldsymbol{\omega}} \tag{2.38a}$$

Hence, given an initial known angular velocity $\boldsymbol{\omega}_0 \in \mathbb{R}^3$, the angular velocity of the body for the current time $t \in \mathbb{R}$ may be computed analytically as

$$\boldsymbol{\omega} = \boldsymbol{\omega}_0 + \int_0^t \boldsymbol{\alpha} dt \tag{2.38b}$$

The ordinary differential equation that relates the body's angular velocity with its orientation quaternion $\hat{\boldsymbol{q}} \in \mathbb{H}$ in space is

$$\frac{d\hat{\boldsymbol{q}}}{dt} = \dot{\boldsymbol{q}} = \frac{1}{2}(0, \boldsymbol{\omega})\hat{\boldsymbol{q}} \tag{2.39}$$

The derivation of the above equation may be obtained by writing the orientation quaternion in the form

$$\hat{\boldsymbol{q}}(t) = \left(\cos\frac{\omega t}{2}, \hat{\boldsymbol{n}}\sin\frac{\omega t}{2}\right) \tag{2.40}$$

where the angular velocity vector $\boldsymbol{\omega} = \omega\hat{\boldsymbol{n}}$ is given in terms of the unit vector $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ representing the axis of rotation and angular speed $\omega \in \mathbb{R}$.

Differentiating $\hat{\boldsymbol{q}}(t)$ with respect to time t using (A.12) results in an equation for the rate of change $\dot{\boldsymbol{q}} \in \mathbb{H}$ of the orientation quaternion

$$\dot{\boldsymbol{q}} = \frac{\omega}{2}\left(-\sin\frac{\omega t}{2}, \hat{\boldsymbol{n}}\cos\frac{\omega t}{2}\right)$$

Evaluating the right hand side of (2.39) using quaternion multiplication (A.5b) yields

$$\frac{1}{2}(0,\boldsymbol{\omega})\hat{\boldsymbol{q}} = \frac{1}{2}\left(-\omega\hat{\boldsymbol{n}}\cdot\hat{\boldsymbol{n}}\sin\frac{\omega t}{2}, \omega\hat{\boldsymbol{n}}\cos\frac{\omega t}{2}\right) = \frac{\omega}{2}\left(-\sin\frac{\omega t}{2}, \hat{\boldsymbol{n}}\cos\frac{\omega t}{2}\right) = \dot{\boldsymbol{q}}$$

Hence, given an initial known orientation $\hat{\boldsymbol{q}}_0 \in \mathbb{H}$, the orientation $\hat{\boldsymbol{q}}$ of the body for the current time $t \in \mathbb{R}$ may be computed analytically in terms of the angular velocity $\boldsymbol{\omega}$ using

$$\hat{\boldsymbol{q}} = \hat{\boldsymbol{q}}_0 + \frac{1}{2}\int_0^t (0,\boldsymbol{\omega})\hat{\boldsymbol{q}}dt \tag{2.41}$$

2.4 Numerical Integration

The ordinary differential equations that describe the motion of bodies in space cannot, in practice, be solved analytically due to the interactive and chaotic nature of simulations within the physics engine. As a motivational example, we consider a computer game where the player must pilot a spacecraft through an asteroid field and battle other computer-controlled craft. During the game, the player controls the craft by switching various thrusters. Likewise, artificial intelligence code controls the hostile agents. Individual asteroids typically tumble freely in space under constant motion but may occasionally collide with each other or any of the spacecraft, resulting in discontinuous reaction forces that alter the objects' paths. In general, the forces applied on bodies vary discontinuously in an arbitrarily complex manner due to human input, autonomous computer agents, collisions and motion constraints imposed between bodies.

The differential equations of motion are thus solved incrementally using *numerical integration*. There exist a number of variants of this technique, each offering a different trade-off between accuracy and computational complexity. The common denominator of all these methods, however, is that the equations are recursively solved for the required values at each simulation step, using computations from earlier steps.

2.4.1 Euler Method

One of the simplest numerical integration techniques is the Euler Method, a first order technique that uses the first two terms from the Taylor expansion of the function [**32**] which, in the context of the physics engine, describes a property of motion with respect to time (Figure 16). Given a function of $f: \mathbb{R} \longmapsto \mathbb{R}^{n}$ of time $t \in \mathbb{R}$ for some $n \in \mathbb{N}$ and some small time delta $\Delta t \in \mathbb{R}$, the Euler method assumes a constant function derivative equal to $f^{'}(t)$ throughout the interval $[t..t+\Delta t]$ and computes an approximation for $f(t+\Delta t)$ as follows

$$f(t+\Delta t) \approx f(t) + \Delta t f^{'}(t) \tag{2.42}$$

This formulation is relatively inexpensive to compute, however, it suffers from stability issues, particularly in the case of *stiff* equations [**33**], such as those describing compressive and tensile forces resulting from very stiff springs, or drag forces induced by a very dense fluid. These issue may be mitigated to a limited extent by using smaller values for Δt.

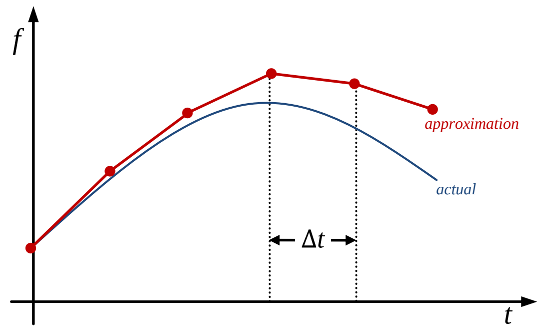


Figure 16 Euler method approximation of a function integral

2.4.2 Verlet Methods

The Verlet family of second-order integration methods [**34**] yield better integral function approximations than the Euler method. The Velocity Verlet uses the first three terms of the Taylor expansion as follows

$$f(t + \Delta t) \approx f(t) + \Delta t f'(t) + \frac{1}{2}\Delta t^2 f''(t) \tag{2.43}$$

Velocity Verlet integration requires the second derivative $f''(t)$ and hence is suitable in cases where this derivative is available, such as when integrating a body's position in terms of its velocity and acceleration

$$\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + \Delta t\, \boldsymbol{v}(t) + \frac{1}{2}\Delta t^2 \boldsymbol{a}(t)$$

Integrating a body's velocity with this method entails the use of the acceleration and its rate of change, sometimes termed *jerk*. However the jerk term is not usually available, in which case, integrating velocity using this method degenerates to the Euler method.

The Velocity Verlet is related to the Basic Verlet integration method, which combines the first three terms from the Taylor expansions of $f(t + \Delta t)$ and $f(t - \Delta t)$ to yield an equation in terms of the previous and the current function value, and the current second derivative, while eliminating the first derivative term. The Basic Verlet approximation is

$$f(t + \Delta t) \approx 2f(t) - f(t - \Delta t) + \frac{1}{2}\Delta t^2 f''(t) \tag{2.44}$$

This formulation is not valid for the first computation at $t = 0$ due to the $f(t - \Delta t)$ term being undefined for $t < \Delta t$. This is solved by using the Velocity Verlet equation for $t = 0$

$$f(\Delta t) \approx f(0) + \Delta t f'(0) + \frac{1}{2}\Delta t^2 f''(0)$$

2.4.3 Fourth-Order Runge-Kutta Method

A more advanced numerical integration method is the fourth-order Runge-Kutta (RK4) method [**35**], belonging to an important family of numerical integration methods bearing the same name. The RK4 method requires the use of the first order derivative of the function, itself a function of time and potentially, of the original function itself

$$f(t + \Delta t) \approx f(t) + \frac{\Delta t}{6}(d_1 + 2d_2 + 2d_3 + d_4) \tag{2.45a}$$

where

$$d_1 = f'\big(t, f(t)\big) \tag{2.45b}$$

$$d_2 = f'\left(t + \frac{\Delta t}{2}, f(t) + \frac{\Delta t}{2} d_1\right) \tag{2.45c}$$

$$d_3 = f'\left(t + \frac{\Delta t}{2}, f(t) + \frac{\Delta t}{2} d_2\right) \tag{2.45d}$$

$$d_4 = f'(t + \Delta t, f(t) + \Delta t d_3) \tag{2.45e}$$

The RK4 method essentially computes a weighted average of four first order derivatives sampled at different times and function approximations within the interval $[t..t + \Delta t]$, giving higher priority to the derivatives computed at the interval midpoint $t + \frac{\Delta t}{2}$. If f' depends only on t, the formulation simplifies to

$$f(t + \Delta t) \approx f(t) + \frac{\Delta t}{6}\left(f'(t) + 4f'\left(t + \frac{\Delta t}{2}\right) + f'(t + \Delta t)\right) \tag{2.46}$$

The last formulation is suitable for use within the physics engine as the dependency of the derivative f' on f is not exposed by the engine's force interface. If f' is constant throughout the interval, the formulation degenerates to the Euler method.

2.5 Collision Detection

In this section we cover the mathematics of collision detection queries, focusing mostly on mid-phase and narrow-phase collision detection. The three fundamental types of geometrical queries are investigated in the following sub-sections.

2.5.1 Intersection Tests

Informally, a test for intersection of two bodies entails determining whether the intersection of the bodies' geometry is an empty set, indicating separation, or if it contains at least one point, indicating contact, or possibly interpenetration (Figure 17). Given the geometry$\chi_1 \subset \mathbb{R}^3$ and $\chi_2 \subset \mathbb{R}^3$ of two bodies expressed as typically continuous point sets in world coordinates, the intersection test may be formally defined as a predicate function $p: \mathcal{P}(\mathbb{R}^3) \times \mathcal{P}(\mathbb{R}^3) \longmapsto \mathbb{B}$ such that

$$p(\chi_1, \chi_2) \stackrel{\text{def}}{=} \chi_1 \cap \chi_2 \neq \emptyset \tag{2.47}$$

where the right-hand expression denotes a Boolean expression whose value may itself be truth T or falsity F.

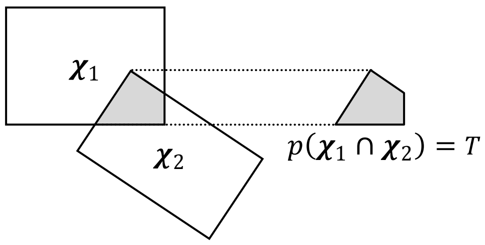


Figure 17 Predicate function for intersection testing

As this predicate function determines only truth or falsity of intersection, the explicit computation of the intersection geometry itself is not required in practice. Custom intersection tests are typically devised for each specific pair of geometry types. In the case when the enclosed point sets are described by simple geometry, the predicate may often be expressed as a simple mathematical inequality, or set of inequalities, involving the parameters of the geometry.

As an example, consider testing intersection for two spheres given by centres $\boldsymbol{c}_1, \boldsymbol{c}_2 \in \mathbb{R}^3$ in world coordinates and radii $r_1, r_2 \in \mathbb{R}$. The predicate may be formulated as

$$p(\boldsymbol{c}_1, r_1, \boldsymbol{c}_2, r_2) \stackrel{\text{def}}{=} |\boldsymbol{c}_2 - \boldsymbol{c}_1| \leq r_1 + r_2 \tag{2.48}$$

which informally states that two spheres are intersecting if the distance between their centres is less than or equal to the sum of their radii (Figure 18).

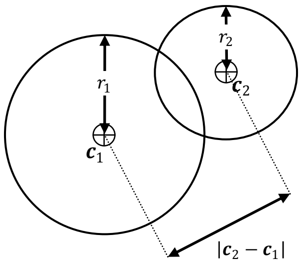


Figure 18 Sphere-to-sphere intersection testing

This formulation requires computation of the magnitude for the offset vector $\boldsymbol{c}_2 - \boldsymbol{c}_1$ which entails a relatively costly square root operation. A more computationally efficient form of the predicate considers the squared distances by using the dot product of the offset vector with itself

$$p(\boldsymbol{c}_1, r_1, \boldsymbol{c}_2, r_2) \stackrel{\text{def}}{=} (\boldsymbol{c}_2 - \boldsymbol{c}_1)^2 \leq (r_1 + r_2)^2 \tag{2.49}$$

Geometrical intersection tests may generally be simplified by operating in the frame of reference of one of the bodies. The motivation for the choice of frame of reference stems from the fact that geometry is typically defined in local coordinates, aligned with the local axes and with the centroid coincident with the local origin. This allows the geometry of both bodies to be specified for testing using fewer parameters, sufficient to describe their shape, provided that an additional transform parameter, such as (2.2), (2.5) or (2.11), be given to specify the relative placement of the geometry of one body with respect to the other. Thus, only one of the geometrical structures needs to be transformed. In addition, simpler tests may be performed thanks to the assumed placement of one of the bodies in its local frame of reference. Formally, the *relative* predicate p_r may be expressed in terms of (2.47) as

$$p_r(\boldsymbol{\chi}_1, \boldsymbol{\chi}_2, \tau_s) \stackrel{\text{def}}{=} p\big(\boldsymbol{\chi}_1, \tau_s(\boldsymbol{\chi}_2)\big) \tag{2.50}$$

where $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$ are point sets enclosed by the geometry in the corresponding body frames of reference, and the transform τ_s is assumed to be the set equivalent of a point-wise world transform τ specifying the relative placement of the second body with respect to the first, defined by

$$\tau_s(\boldsymbol{\chi}) = \{\tau(x) | x \in \boldsymbol{\chi}\} \tag{2.51}$$

The transformation τ_s effectively brings the point set $\boldsymbol{\chi}_2$ into the frame of reference of $\boldsymbol{\chi}_1$, allowing the predicate p to operate in a common frame of reference.

Revisiting the sphere-to-sphere intersection test, the geometry may be expressed simply in terms of the radii r_1 and r_2, where the centroid of each sphere is assumed to coincide with the CM and lying on the origin of the corresponding body frame of reference. By assuming a quaternion-based transform $\tau = (\hat{\boldsymbol{q}}, \boldsymbol{t})$, the relative intersection predicate may be expressed as

$$p_r(r_1, r_2, \tau) \stackrel{\text{def}}{=} \boldsymbol{t}^2 \leq (r_1 + r_2)^2 \tag{2.52}$$

As the shape of a sphere is invariant under rotation, the predicate p_r depends only on the translation component $\boldsymbol{t} \in \mathbb{R}^3$ of τ. Hence, the test simplifies to a simple comparison of the squared translation distance against the squared sum of the sphere radii.

When intersection testing geometry of different types, the test is generally simplified the most by operating in the frame of reference of the most complex body. When testing, for instance, a box against a sphere, the latter is transformed into the frame of reference of the former. The test thus consists in computing the closest point, within the box, to the sphere centroid [**36**], and determining if the distance from this point to this centroid is within the radius of the sphere (Figure 19). Incidentally, this technique may be applied for testing any type of geometry with a sphere, provided that a means for computing the closest point within the geometry to a given internal or external point is available.

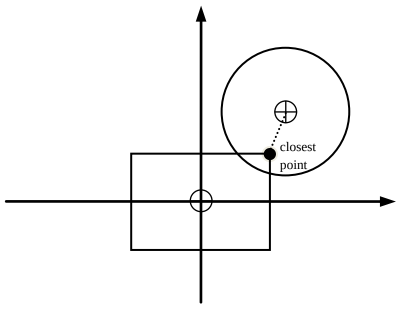


Figure 19 Box-to-sphere intersection testing

Researchers have devised a number of intersection testing techniques, specific to certain types of geometry, or of more generic application. For reasons of brevity, we provide a brief outline with appropriate references for supplemental reading.

2.5.1.1 The method of Separating Axes

The method of separating axes [**12**], also known as the *Separating Axis Test* (SAT), operates on the fact that a separating plane can be found between any pair of non-intersecting convex shapes. Hence, it is possible to identify an axis, perpendicular to this plane, such that the projection of the shapes on to this axis yields two non-intersecting intervals (Figure 20). Although an infinite continuum of such axes may exist, for specific cases it is possible to enumerate and test a limited set. For instance, when the geometry being tested is polyhedral in nature, it is sufficient to test the face normals of both polyhedra and the vector-products of all the edge pairs, taken respectively from the two polyhedra. The latter tests are important in situations where the closest features of the polyhedra are edges such that all the face normal tests fail to identify separation. For geometry with curved surfaces such as spheres, cones or cylinders, candidate normals must be chosen carefully such that they point in an *appropriate* direction towards the other geometry involved in the test.

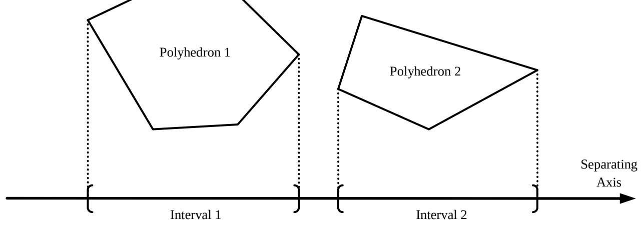


Figure 20 The method of separating axes

In set-theory, projecting a geometric volume onto an axis entails computing the real interval consisting of the scalar product of all position vectors of the points within the geometry, with the axis vector. Given axis $\boldsymbol{a} \in \mathbb{R}^3$ and the point-set $\boldsymbol{\chi} \subset \mathbb{R}^3$ of the geometric volume, the projection $\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}} \subset \mathbb{R}$ is defined by

$$\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}} \stackrel{\text{def}}{=} \{\boldsymbol{x} \cdot \boldsymbol{a} | \boldsymbol{x} \in \boldsymbol{\chi}\} \tag{2.53}$$

As the point-set $\boldsymbol{\chi}$ is continuous in this case, the set $\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}}$ is a continuous closed interval in $\mathbb{R}$. Hence, $\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}}$ may be defined in terms of its lower and upper bound values, that is

$$\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}} \stackrel{\text{def}}{=} [min(\{\boldsymbol{x} \cdot \boldsymbol{a} | \boldsymbol{x} \in \boldsymbol{\chi}\})..max(\{\boldsymbol{x} \cdot \boldsymbol{a} | \boldsymbol{x} \in \boldsymbol{\chi}\})] \tag{2.54}$$

Definition (2.54) facilitates the computation of $\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}}$ as only the lower and upper bound values are required. For instance, the interval $\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}}$ resulting from projecting a sphere centred at $\boldsymbol{c} \in \mathbb{R}^3$, radius $r \in \mathbb{R}$, onto the axis $\boldsymbol{a}$ is given by

$$\boldsymbol{\psi}_{\boldsymbol{a},\boldsymbol{\chi}} \stackrel{\text{def}}{=} [\boldsymbol{c} \cdot \boldsymbol{a} - r, \boldsymbol{c} \cdot \boldsymbol{a} + r]$$

which is easy to verify by inspection.

2.5.1.2 The Gilbert-Johnson-Keerthi Algorithm

A very efficient intersection algorithm for convex polyhedra is the Gilbert-Johnson-Keerthi (GJK) algorithm [**37**,**16**]. The GJK algorithm operates on the Minkowski difference [**38**] of the body geometry $\boldsymbol{\chi}_1 \subset \mathbb{R}^3$ and $\boldsymbol{\chi}_2 \subset \mathbb{R}^3$, denoted by $\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \subset \mathbb{R}^3$, that is defined as

$$\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \stackrel{\text{def}}{=} \{\boldsymbol{x}_2 - \boldsymbol{x}_1 | \boldsymbol{x}_1 \in \boldsymbol{\chi}_1, \boldsymbol{x}_2 \in \boldsymbol{\chi}_2\} \tag{2.55}$$

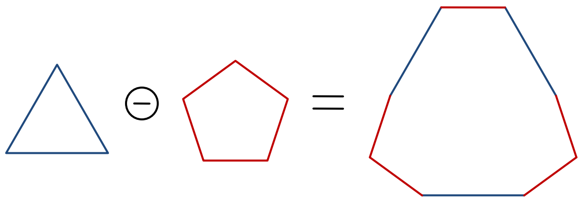


Figure 21 Minkowski difference of two convex regions

The Minkowski difference $\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1$ results in a convex region (Figure 21) that contains the origin if, and only if, the regions $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$ intersect. That is, for some $\boldsymbol{x}_1 \in \boldsymbol{\chi}_1$ and $\boldsymbol{x}_2 \in \boldsymbol{\chi}_2$,

$$\boldsymbol{0} \in \boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1 \Leftrightarrow \boldsymbol{x}_2 - \boldsymbol{x}_1 = \boldsymbol{0} \Leftrightarrow \boldsymbol{x}_1 = \boldsymbol{x}_2 \Leftrightarrow \boldsymbol{\chi}_1 \cap \boldsymbol{\chi}_2 \neq \emptyset$$

If $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$ are polyhedra, $\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1$ is also a polyhedron, and may be constructed from the convex hull of the point cloud generated by subtracting every vertex from $\boldsymbol{\chi}_2$ from every other vertex in $\boldsymbol{\chi}_1$. The GJK algorithm searches for the origin by building internal simplices, consisting of points, lines, triangles or tetrahedrons, using the vertices of $\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1$ and iteratively converging towards the origin using a *support function* until no more progress can be made. Rather than explicitly constructing the set $\boldsymbol{\chi}_2 \ominus \boldsymbol{\chi}_1$, GJK, samples its vertices by applying the support function on the vertices of $\boldsymbol{\chi}_1$ and $\boldsymbol{\chi}_2$ using the linearity property of this function.

If the GJK algorithm fails to converge to the origin, the final approximation is its closest point whose magnitude as a position vector yields a separation distance. Thus, a simplified form of GJK, that operates on the vertices of a convex region $\boldsymbol{\chi} \subset \mathbb{R}^3$, may be used to compute the closest point within $\boldsymbol{\chi}$, to some given internal or external point $\boldsymbol{p} \in \mathbb{R}^3$. This finds application in convex polyhedron-to-sphere intersection tests, whereby the distance of the closest point of a sphere centre to the convex polyhedron is computed and compared with the sphere radius.

2.5.2 Time of Impact Estimation

Impact estimation queries are inherently more complex than interference tests as the time dimension must be factored into the computations. Due to the chaotic and unpredictable nature of interactive simulations, the possibility of impact and TOI cannot be accurately determined. However, a number of simplifications are often assumed, mainly:

- The TOI test is limited to a small time interval, typically the maximum time step of the simulation, such that the current body trajectories may be assumed linear or ballistic in the short term
- Constant linear and angular velocity is assumed to simplify computations
- Angular velocity is assumed zero for the duration of the interval, enabling the test to deal with translational motion only

The justification for these assumptions follows from the notion that the bodies' kinetic properties vary only slightly over short time intervals. Thus, a sufficiently adequate estimate for the TOI may still be computed under these assumptions.

The concept of operating in the frame of reference of one of the bodies being tested, as embodied in (2.48), may also be applied for TOI estimation to simplify computations by assuming the second body moving with respect to the first body lying stationary at the local origin. Therefore, the position, orientation, and the linear and angular components of velocity are computed for the second body with respect to the first. The TOI test thus only needs to determine whether the volume swept by the trajectory of the moving body intersects with the geometry of the stationary body.

Given the kinetic properties $\boldsymbol{K}_1(t) = (\boldsymbol{x}_1, \hat{\boldsymbol{q}}_1, \boldsymbol{v}_1, \boldsymbol{\omega}_1)$ and $\boldsymbol{K}_2(t) = (\boldsymbol{x}_2, \hat{\boldsymbol{q}}_2, \boldsymbol{v}_2, \boldsymbol{\omega}_2)$ of two bodies being tested, where $\boldsymbol{x}_i(t) \in \mathbb{R}^3$, $\hat{\boldsymbol{q}}_i(t) \in \mathbb{H}$, $\boldsymbol{v}_i(t) \in \mathbb{R}^3$ and $\boldsymbol{\omega}_i(t) \in \mathbb{R}^3$ are, respectively, the position vector, orientation quaternion, linear velocity and angular velocity of the bodies at time t, the relative kinetic properties of the second body with respect to the first are given by

$$\boldsymbol{K}_{\mathrm{r}}(t) = (\boldsymbol{x}_2 - \boldsymbol{x}_1, \hat{\boldsymbol{q}}_1{}^{*}\hat{\boldsymbol{q}}_2, \boldsymbol{v}_2 - \boldsymbol{v}_1, \boldsymbol{\omega}_2 - \boldsymbol{\omega}_1) \tag{2.56}$$

2.5.2.1 Sphere-to-Sphere Test

As an example, we consider the TOI estimation test for two spheres with radii $r_1, r_2 \in \mathbb{R}$, relative position $\boldsymbol{c}_\mathrm{r} \in \mathbb{R}^3$, and relative linear velocity $\boldsymbol{v}_\mathrm{r} \in \mathbb{R}^3$ of the second sphere with respect to the first (Figure 22). For spheres, the relative angular velocity may be ignored as their geometry is invariant under rotation. By further assuming constant linear velocity, the equation of relative motion for the second sphere centroid may be represented as a linear function of relative time t, where $t = 0$ corresponds to the current simulation time, given by

$$\boldsymbol{c}_\mathrm{r}(t) = \boldsymbol{c}_0 + t\boldsymbol{v}_\mathrm{r} \tag{2.57}$$

where $\boldsymbol{c}_0 = \boldsymbol{c}_\mathrm{r}(0)$. The second sphere makes contact with the first when $\boldsymbol{c}_\mathrm{r}{}^2(t) = (r_1 + r_2)^2$. Expanding the definition of $\boldsymbol{c}_r(t)$ yields a quadratic equation in t of the form

$$(\boldsymbol{v}_\mathrm{r}{}^2)t^2 + (2\boldsymbol{c}_0\boldsymbol{v}_\mathrm{r})t + (\boldsymbol{c}_0{}^2 - (r_1 + r_2)^2) = 0$$

The two roots of this equation correspond to the times of entry and exit of the second sphere into and out of the first. If the equation has no real roots, then the second sphere never intersects the first along the line of motion and impact is ruled out under the given assumptions. Otherwise, both roots are real with values $t_\mathrm{min}, t_\mathrm{max} \in \mathbb{R}$ such that $t_{min} \leq t_{max}$. The root of interest is the one corresponding to the time of entry. If $0 \leq t_\mathrm{min} \leq t_\mathrm{max}$, the interpenetration occurs forward in time, and hence t_min gives the required TOI, relative to the current time. If $t_\mathrm{min} \leq t_\mathrm{max} < 0$, the interpenetration time interval occurs completely in the past along the assumed line of motion, and hence impact is be ruled out. If $t_\mathrm{min} < 0 \leq t_\mathrm{max}$, the spheres are currently interpenetrating, in which case the current time $t = 0$ may be assumed as the TOI. The special case where $0 \leq t_\mathrm{min} = t_\mathrm{max}$ indicates the case when the second sphere comes into tangential contact with the first at a future instant in time, and may safely be ignored as a collision. The estimated points of entry and exit $\boldsymbol{p}_\mathrm{min}, \boldsymbol{p}_\mathrm{max} \in \mathbb{R}^3$ may be computed as $\boldsymbol{p}_\mathrm{min} = \boldsymbol{c}_0 + t_\mathrm{min}\boldsymbol{v}_\mathrm{r}$ and $\boldsymbol{p}_\mathrm{max} = \boldsymbol{c}_0 + t_\mathrm{max}\boldsymbol{v}_\mathrm{r}$.

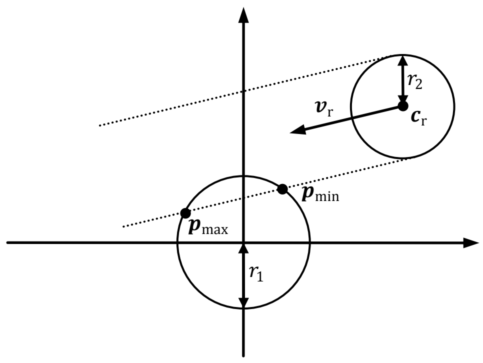

r_2
$\boldsymbol{v}_\mathrm{r}$
$\boldsymbol{c}_\mathrm{r}$
$\boldsymbol{p}_\mathrm{min}$
$\boldsymbol{p}_\mathrm{max}$
r_1

Figure 22 TOI estimation between two spheres

2.5.2.2 Sphere-to-Half-Space Test

A half-space is defined as one of the two unbounded regions resulting from partitioning $\mathbb{R}^3$ by a plane, where we adopt the convention of the boundary plane normal directed outwards. Formally, a half-space may be defined by $\boldsymbol{r} \cdot \hat{\boldsymbol{n}} \leq d$ where $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ is a unit vector defining outward plane boundary normal, $d \in \mathbb{R}$ is the signed distance from the origin, and $\boldsymbol{r} \in \mathbb{R}^3$ is any point within the half-space or on its boundary.

A derivation for estimating the TOI of a sphere against a half-space (Figure 23) follows a similar approach to the sphere-to-sphere test. The test assumes operation in the half-space's frame of reference and hence its boundary plane is represented by the Cartesian equation $y = 0$. Thus the boundary contains the origin and its normal $\hat{\boldsymbol{n}} = [0 \quad 1 \quad 0]$ is aligned with the y-axis, such that any point $\boldsymbol{r}$ on the plane satisfies $\boldsymbol{r} \cdot [0 \quad 1 \quad 0] \leq 0$. The signed distance of the relatively moving sphere centre $\boldsymbol{c}_r(t) \in \mathbb{R}^3$ relative to the half-space is thus given by $d(t) = \boldsymbol{c}_r \cdot \hat{\boldsymbol{n}} \in \mathbb{R}$ which is effectively the y-component of $\boldsymbol{c}_r$. Impact occurs when $(t) = r$, where $r \in \mathbb{R}$ is the sphere radius. Expanding $d(t) = r$ using (2.57) yields a linear equation in t whose root gives the TOI t_{impact} as follows

$$(\boldsymbol{c}_0 + t\boldsymbol{v}_{\text{r}}) \cdot \hat{\boldsymbol{n}} = r$$

and hence

$$t_{\text{impact}} = \frac{r - \boldsymbol{c}_0 \cdot \hat{\boldsymbol{n}}}{\boldsymbol{v}_{\text{r}} \cdot \hat{\boldsymbol{n}}} = \frac{r - \text{c}_0^y}{\text{v}_{\text{r}}^y}$$

where $\text{c}_0^y \in \mathbb{R}$ and $\text{v}_{\text{r}}^y \in \mathbb{R}$ are, respectively, the y-component of the relative sphere centre at relative time zero $\boldsymbol{c}_0$ and the y-component of the relative sphere velocity $\boldsymbol{v}_{\text{r}}$. If t_{impact} is negative, the impact occurs in the relative past under the assumed motion and hence indicates separation, otherwise, its value gives the relative time of impact with respect to the current simulation time. Given t_{impact}, the TOI $\boldsymbol{p}_{\text{impact}} \in \mathbb{R}^3$ may be computed using $\boldsymbol{p}_{\text{impact}} = \boldsymbol{c}_0 + t_{\text{impact}}\, \boldsymbol{v}_{\text{r}}$.

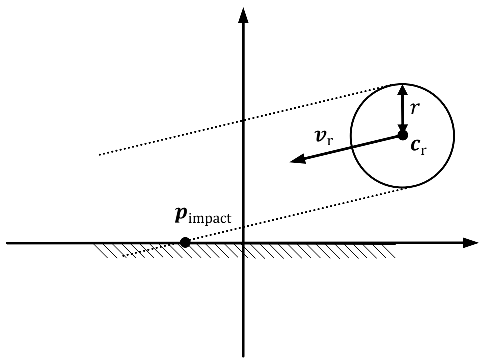

r
$\boldsymbol{v}_{\text{r}}$
$\boldsymbol{c}_{\text{r}}$
$\boldsymbol{p}_{\text{impact}}$

Figure 23 TOI estimation between a sphere and a half-space

Care should be taken to avoid computing the equation for t_{impact} when $\text{v}_{\text{r}}^y \approx 0$, which may result in a numerical overflow error. This condition indicates that the sphere is moving parallel to the half-space boundary and thus cannot intercept it, unless $c_0^y \leq r$, in which case that the sphere is already intersecting the half-space at $t = 0$.

2.5.2.3 Tests for more Complex Geometry

The sphere-to-sphere and sphere-to-half-space test derivations detailed in (§§ 2.5.2.1, 2.5.2.2) provide an indication of the increased complexity inherent to TOI estimation vis-à-vis testing two geometric volumes for intersection at a given instant in time. For more complex types of geometry, such as convex polyhedra, a number of specialised algorithms have been developed. For reasons of scope and brevity, we have omitted the details of these algorithms from this paper. We encourage the interested reader to study [**39**], [**40**], [**41**] and [**15**].

2.5.3 Contact Manifold Generation

When two simulated bodies are in contact, the respective geometrical features of the bodies are identified for subsequent collision resolution. Depending on the geometry of the bodies, the features that are in contact may consist of one or more points, lines or areas, collectively termed the *contact manifold* (Figure 24). For convex bodies, the contact manifold lies in a *contact plane* with which is associated a *contact normal*, orthogonal to it. This statement is not generally true for concave bodies however, as a concave body may be in contact at multiple, or potentially a continuum of points, each with its own contact normal.

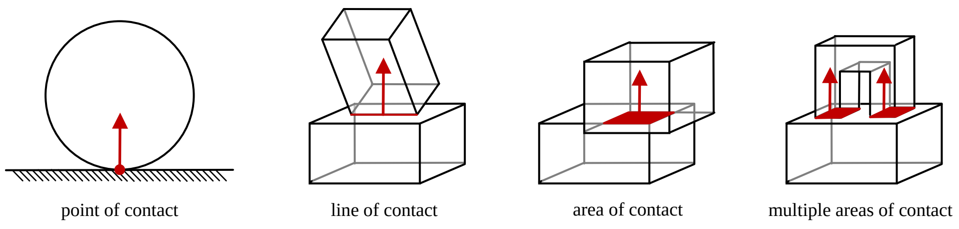


Figure 24 Contact manifolds for various geometry and contact configurations

2.5.3.1 Reduced Contact Manifold

The contact manifold defines areas on the surface of the bodies where contact forces are to be applied. As the contact manifold may contain an infinite number of points, ideally the forces are continuously distributed over the manifold by integrating their derivative pressure functions over the areas defined by the manifold. Such an approach is computationally intractable given real-time requirements. However, by the principle of *contact force and impulse equivalence* [**42**] it is possible to reduce the contact manifold to a finite set of contact points, such that equivalent forces or impulses may be applied to yield the same result. For convex bodies, this discrete point set consists of the vertices of the convex hull enclosing the contact manifold. For concave bodies with multiple regions in contact, each region may be similarly reduced to the vertices of its convex hull (Figure 25). Generating the contact manifold between two bodies in contact entails performing a spatial query whose implementation depends on the type of body geometry. Due to numerical inaccuracy, this query cannot assume that the bodies are exactly in contact as they may also be interpenetrating. Hence, the query must deal with such states appropriately and compute a penetration depth for each contact point that may be used to correct the overall interpenetration between the bodies.

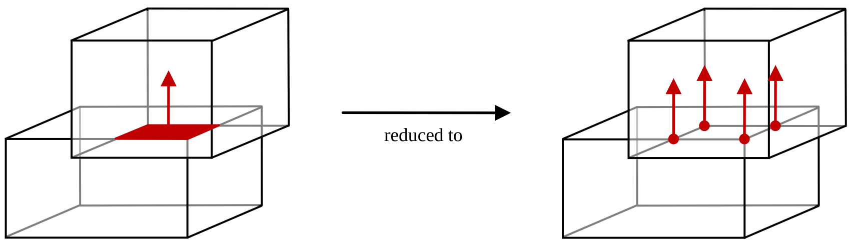


Figure 25 Contact manifold reduction by the principle of contact force and impulse equivalence

A contact occurs whenever a geometrical feature of one body penetrates or comes into contact with a feature of the other. A feature may be a vertex, edge or face, and hence there are six possible feature combinations that may occur, namely: vertex-vertex, vertex-edge, vertex-face, edge-edge, edge-face and face-face (Figure 26).

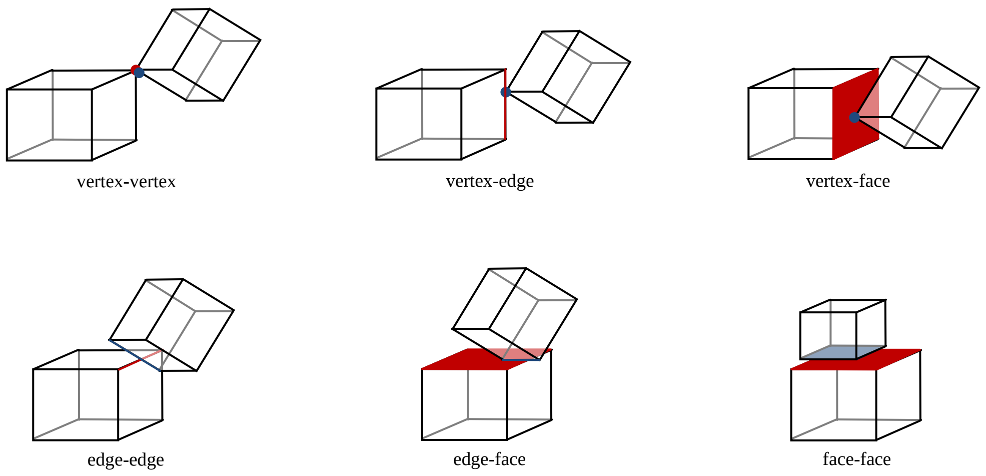


Figure 26 The six geometrical feature contact cases

The most general occurrence is the vertex-face case, whereby a corner of one body comes into contact with a face of the other. In this case, the contact normal assumes the value of the face normal and the penetration depth is computed from the signed perpendicular distance of the vertex under the plane containing the face.

A slightly less common occurrence is the edge-edge case, whereby one body comes into contact with the other edge-wise. In this case, the normal may be computed as a vector perpendicular to both edges by applying the vector product on the corresponding edge vectors. The special case when the edges are parallel cannot be handled in this manner, however, the contact is in this case detected by the vertex-face case.

In practice, a vertex–vertex contact is a very rare occurrence and in the context of limited precision digital computation, detection of such occurrence is impractical, as a distance tolerance factor would have to be applied. It is also difficult to identify a contact normal for two vertices in contact due to their dimensionless nature. When two vertices are in such a configuration, a face adjacent to one of the vertices is invariably penetrated by the other vertex, which is detected by the vertex-face case. Hence, vertex-vertex contacts may be safely ignored. Likewise, the vertex-edge case may also be ignored as it is covered by the vertex-face case where the face is one adjacent to the edge such that it is closest to the vertex.

In static body configurations, the most common feature occurrence tends to be the face-face case, whose intersection for flat faces is a bounded area. The reduced manifold may however by obtained from the vertex-face and edge-edge tests of the adjacent vertices and edges. Thus, the six feature contact cases may be effectively reduced to two: the vertex-face and edge-edge case. It should be noted that when one or more of the body surfaces is curved, such as a sphere resting on a ground plane, the resulting manifold is still a single point.

As with intersection and TOI estimation tests, the computation of a contact manifold may be greatly simplified by operating in the frame of reference of one of the bodies, preferably the one with the most complex geometry, as eventually demonstrated by the examples further down this section.

Formally, a reduced contact manifold of n points may be defined as a vector

$$\boldsymbol{m} = [\boldsymbol{c}_1 \quad \boldsymbol{c}_2 \quad \dots \quad \boldsymbol{c}_n] \in \mathbb{R}^{n\times(3+3+1)} \tag{2.58a}$$

of contact points $\boldsymbol{c}_i \in \mathbb{R}^{3+3+1}$. Each contact point may in turn be defined by the triple

$$\boldsymbol{c}_i = (\boldsymbol{p}_i, \hat{\boldsymbol{n}}_i, d_i) \tag{2.58b}$$

such that $\boldsymbol{p}_i \in \mathbb{R}^3$ is the position, $\hat{\boldsymbol{n}}_i \in \mathbb{R}^3$ is a unit vector representing the contact normal, and $d_i \in \mathbb{R}$ is the interpenetration depth at the ith contact point. A convention for the direction of $\hat{\boldsymbol{n}}_i$ is usually adopted, such as ensuring that the normal points away from the feature of the first body.

2.5.3.2 Manifold for Sphere-to-Sphere Contact

As an example, we consider generation of a contact manifold of two spheres in contact, or slightly interpenetrating, with the centroid of the second sphere positioned $\boldsymbol{c}_r \in \mathbb{R}^3$ relative to the first and with radii $r_1, r_2 \in \mathbb{R}$.

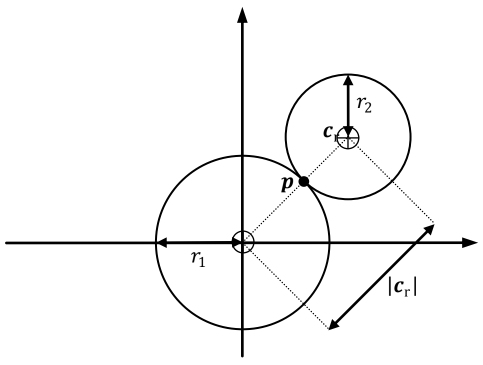


Figure 27 Two spheres in contact

The two spheres in ideal contact (Figure 27) meet at a single point $\boldsymbol{p} \in \mathbb{R}^3$ lying on the line joining the sphere centres from the origin to point $\boldsymbol{c}_r$ where $|\boldsymbol{c}_r| = r_1 + r_2$. The contact point $\boldsymbol{p}$ may thus be computed using

$$\boldsymbol{p} = \frac{r_1}{r_1 + r_2}\boldsymbol{c}_r$$

In practice, the more general condition $|\boldsymbol{c}_r| \leq r_1 + r_2$ applies due to interpenetration, resulting in a *lentiform* intersection volume containing infinitely many points. Assuming the sphere regions are given in the frame of reference of the first sphere by the point sets $\boldsymbol{\chi}_1 = \{\boldsymbol{x} \in \mathbb{R}^3 | r_1 \geq |\boldsymbol{x}|\}$ and $\boldsymbol{\chi}_2 = \{\boldsymbol{x} \in \mathbb{R}^3 | r_2 \geq |\boldsymbol{x} - \boldsymbol{c}_r|\}$, then

$$|\boldsymbol{c}_r| \leq r_1 + r_2 \Leftrightarrow \frac{r_1}{r_1 + r_2}|\boldsymbol{c}_r| \leq r_1 \Leftrightarrow |\boldsymbol{p}| \leq r_1 \Leftrightarrow \boldsymbol{p} \in \boldsymbol{\chi}_1$$

Similarly,

$$|\boldsymbol{c}_r| \leq r_1 + r_2 \Leftrightarrow \left(\frac{r_2}{r_1 + r_2}\right)|\boldsymbol{c}_r| \leq r_2 \Leftrightarrow \left|\frac{r_1}{r_1 + r_2}\boldsymbol{c}_r - \boldsymbol{c}_r\right| \leq r_2 \Leftrightarrow |\boldsymbol{p} - \boldsymbol{c}_r| \leq r_2 \Leftrightarrow \boldsymbol{p} \in \boldsymbol{\chi}_2$$

Therefore $\boldsymbol{p} \in \boldsymbol{\chi}_1 \cap \boldsymbol{\chi}_2$ and thus the computation for $\boldsymbol{p}$ still provides an acceptable value in situations of interpenetration, whereby $\boldsymbol{p}$ is contained within the lentiform $\boldsymbol{\chi}_1 \cap \boldsymbol{\chi}_2$ of the two spheres.

By keeping to the convention of the contact normal $\hat{\boldsymbol{n}}$ pointing away from the first sphere, $\hat{\boldsymbol{n}}$ may be computed very simply using $\hat{\boldsymbol{n}} = \frac{\boldsymbol{c}_r}{|\boldsymbol{c}_r|}$. The penetration depth d may be computed as $d = r_1 + r_1 - |\boldsymbol{c}_r|$.

2.5.3.3 Manifold for Box-to-Half-space Contact

We consider a second manifold generation example involving a box against a half-space (Figure 28). By working in the frame of reference of the half-space, the boundary plane is represented by Cartesian equation $y = 0$ and associated normal $\hat{\boldsymbol{n}} = [0 \quad 1 \quad 0]$. The box is thus given in terms of a relative position for its centroid and a relative orientation for its geometry.

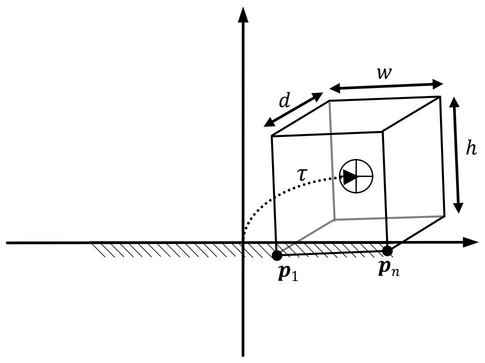


Figure 28 A box in contact with a half-space

Collectively, these placement parameters are embodied in an affine transform $\tau\colon \mathbb{R}^3 \longmapsto \mathbb{R}^3$, such as (2.11). The box dimensions may be represented by $w, h, d \in \mathbb{R}$. In the frame of reference of the box, the vertices consist of the set

$$\boldsymbol{V} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \middle| \boldsymbol{v} = \frac{1}{2}[\pm w \quad \pm h \quad \pm d] \right\}$$

The vertex set $\boldsymbol{V}$ is transformed into the half-space's frame of reference using τ to yield a new set $\boldsymbol{V}_\mathrm{r} = \{\tau(\boldsymbol{v}) | \boldsymbol{v} \in \boldsymbol{V}\}$. The box is in contact or interpenetrating the half-space if at least one vertex is below the boundary plane. The set of such vertices is given by $\boldsymbol{P} = \{\boldsymbol{p} \in \boldsymbol{V}_\mathrm{r} | \boldsymbol{p} \cdot \hat{\boldsymbol{n}} \leq 0\}$ whose elements are enumerated by $\boldsymbol{p}_i$ for $1 \leq i \leq m$ where $m \in \mathcal{N}$ is the size of the set $\boldsymbol{P}$. The points $\boldsymbol{p}_i$ are effectively the positions of the contact points for the reduced manifold. The ith contact normal $\hat{\boldsymbol{n}}_i$ is common to all contact points such that $\hat{\boldsymbol{n}}_i = \hat{\boldsymbol{n}}$. The ith penetration depth d_i is the distance of $\boldsymbol{p}_i$ from the plane boundary, computed as $d_i = -\boldsymbol{p}_i \cdot \hat{\boldsymbol{n}}$.

2.5.3.4 Manifold Generation for more Complex Geometry

Contact manifold generation queries share many techniques with geometric intersection tests. For reasons of scope and brevity, once again we omit further examples involving more complex geometry. However, some relevant techniques will be summarised, with references provided for supplemental reading.

For convex geometry, a variant of SAT may be used [**43**]. This method searches for the axis of least overlap by iteratively testing a discrete number of candidate axes. Once such an axis is identified, the closest features of the geometry are determined using a support function that projects the features onto the axis. The reduced contact manifold is thus generated from the vertices constituting the chosen features. Using this technique the contact normal is usually derived from the axis of least overlap or from the contact feature with highest dimension.

Another technique, focusing on the computation of the penetration depth of two convex interpenetrating bodies, entails the use of a variation [**44**], [**45**] of the GJK algorithm. The technique relies on the fact that the Minkowski difference of the body geometry results in a region containing the origin, by virtue of interpenetration of the bodies. The least distance of the origin from the boundary of this region is effectively the penetration depth. As with GJK, the Minkowski difference is sampled, rather than computed explicitly, and the algorithm progressively converges to the solution using a support function. The algorithm may in some cases be unable to compute an exact solution and hence it iterates until some tolerance criterion is met.

2.6 Collision Response

In this section we cover the principles of collision resolution in rigid body dynamics and describe a contact model together with a number of approaches for resolving collisions.

Two rigid bodies in unconstrained motion, potentially under the action of forces, may be modelled by solving their equations of motion described in (§ 2.3) using numerical integration techniques such as those described in (§ 2.4). On collision, the kinetic properties of two such bodies seem to undergo an instantaneous change, typically resulting in the bodies rebounding away from each other, sliding, or settling into relative static contact, depending on the elasticity of the materials and the configuration of the collision.

2.6.1 Contact Forces

The origin of the rebound phenomenon, or *reaction*, may be traced to the behaviour of real bodies that, unlike their perfectly rigid simulated counterparts, do undergo minor compression on collision, followed by expansion, prior to separation. The compression phase converts the kinetic energy of the bodies into potential energy and to an extent, heat. The expansion phase converts the potential energy back to kinetic energy.

2.6.1.1 Reaction

During the compression and expansion phases of two colliding bodies, each body generates reactive forces on the other at the points of contact, such that the sum reaction forces of one body are equal in magnitude but opposite in direction to the forces of the other, as per the Newtonian principle of action and reaction. It should be noted that, ignoring the effects of friction, a collision affects only the component of the velocities along the contact normal, leaving the tangential components unaffected (Figure 29).

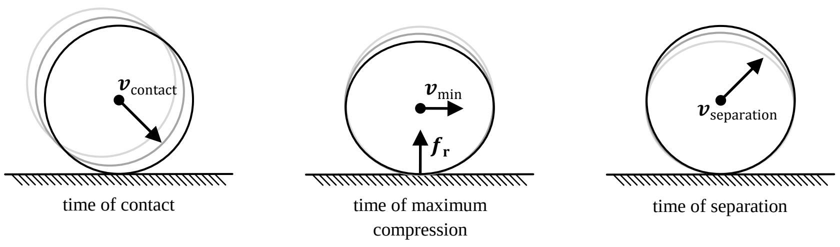

$\boldsymbol{v}_{\text{contact}}$
$\boldsymbol{v}_{\text{min}}$
$\boldsymbol{f}_{\text{r}}$
$\boldsymbol{v}_{\text{separation}}$
time of contact
time of maximum compression
time of separation

Figure 29 The compression and expansion phases of collision

The degree of relative kinetic energy retained after a collision, termed the *restitution*, is dependent on the elasticity of the bodies' materials. The *coefficient of restitution* $e \in \mathbb{R}$ between two given materials is measured as the ratio of the relative post-collision speed of a point of contact along the contact normal, with respect to the relative pre-collision speed of the same point along the same normal. It should be noted that the kinetic energy loss is relative to one body with respect to the other. Thus the total momentum of both bodies with respect to some common reference is unchanged after the collision, in line with the principle of *conservation of momentum*.

2.6.1.2 Surface-to-Surface Friction

Another important contact phenomenon is surface-to-surface friction, a force that impedes the relative motion of two surfaces in contact, or that of a body in a fluid. In this section we discuss surface-to-surface friction of two bodies in relative static contact or sliding contact. In the real world, friction is due to the imperfect microstructure of surfaces whose protrusions interlock into each other (Figure 30) generating reactive forces tangential to the surfaces.

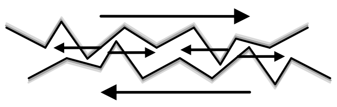


Figure 30 Friction due to surface microstructure imperfections

To overcome the friction between two bodies in static contact, the surfaces must somehow lift away from each other. Once in motion, the degree of surface affinity is reduced and hence bodies in sliding motion tend to offer lesser resistance to motion. These two categories of friction are respectively termed *static friction* and *dynamic friction*.

2.6.2 Impulse-based Contact Model

A force $\boldsymbol{f}(t) \in \mathbb{R}^3$ acting on a body of mass $m \in \mathbb{R}$ for a time interval $[t_0 .. t_1]$ generates a change in the body's momentum $\boldsymbol{p}(t) = m\boldsymbol{v}(t)$. The change in momentum, termed an *impulse* and denoted by $\boldsymbol{j} \in \mathbb{R}^3$ is thus computed as

$$\boldsymbol{j} = \int_{t_0}^{t_1} \boldsymbol{f} dt \tag{2.59}$$

For fixed $\boldsymbol{j}$, the equation suggests that $t_1 \to t_0 \Rightarrow |\boldsymbol{f}| \to \infty$, that is, a smaller time interval must be compensated by a stronger reaction force to achieve the same impulse. When modelling a collision between idealized rigid bodies, it is impractical to simulate the compression and expansion phases of the body geometry over the collision time interval. However, by assuming that a suitable $\boldsymbol{f}$ can be found such that the limit

$$\lim_{t_1 \to t_0} \int_{t_0}^{t_1} \boldsymbol{f} dt$$

exists and is equal to $\boldsymbol{j}$, the notion of *instantaneous impulses* may be introduced to simulate an instantaneous change in velocity after a collision.

2.6.2.1 Impulse-based Reaction

The effect of the reaction force $\boldsymbol{f}_{\mathrm{r}}(t) \in \mathbb{R}^3$ over the interval of collision $[t_0 .. t_1]$ may hence be represented by an instantaneous reaction impulse $\boldsymbol{j}_{\mathrm{r}} \in \mathbb{R}^3$, computed as

$$\boldsymbol{j}_{\mathrm{r}} = \int_{t_0}^{t_1} \boldsymbol{f}_{\mathrm{r}} dt$$

By deduction from the principle of action and reaction, if the collision impulse applied by the first body on the second body at a contact point $\boldsymbol{p} \in \mathbb{R}^3$ is $\boldsymbol{j}_{\mathrm{r}}$, the counter impulse applied by the second body on the first is $-\boldsymbol{j}_{\mathrm{r}}$ (Figure 31). The decomposition $\pm\boldsymbol{j}_{\mathrm{r}} = \pm j_{\mathrm{r}}\hat{\boldsymbol{n}}$ into the impulse magnitude $j_{\mathrm{r}} \in \mathbb{R}$ and direction along the contact normal $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ and its inverse $-\hat{\boldsymbol{n}}$ allows for the derivation of a formula to compute the change in linear and angular velocities of the bodies resulting from the collision impulses.

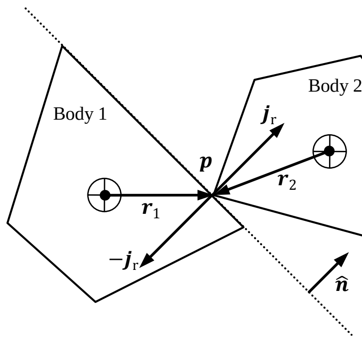


Figure 31 The application of impulses at the point of collision

Assuming the collision impulse magnitude j_{r} is given, the change in the bodies' linear velocities is computed as follows

$$\boldsymbol{v}_1' = \boldsymbol{v}_1 - \frac{j_{\mathrm{r}}}{m_1}\hat{\boldsymbol{n}} \tag{2.60a}$$

$$\boldsymbol{v}_2' = \boldsymbol{v}_2 + \frac{j_\mathrm{r}}{m_2}\hat{\boldsymbol{n}} \tag{2.60b}$$

where, for the ith body, $\boldsymbol{v}_i \in \mathbb{R}^3$ is the linear pre-collision velocity, $\boldsymbol{v}_i' \in \mathbb{R}^3$ is the linear post-collision velocity and $m_i \in \mathbb{R}$ is the mass.

Similarly, the change in angular velocities due to the impulses is

$$\boldsymbol{\omega}_1' = \boldsymbol{\omega}_1 - j_\mathrm{r}\boldsymbol{I}_1{}^{-1}(\boldsymbol{r}_1 \times \hat{\boldsymbol{n}}) \tag{2.61a}$$

$$\boldsymbol{\omega}_2' = \boldsymbol{\omega}_2 + j_\mathrm{r}\boldsymbol{I}_2{}^{-1}(\boldsymbol{r}_2 \times \hat{\boldsymbol{n}}) \tag{2.61b}$$

where, for the ith body, $\boldsymbol{\omega}_i \in \mathbb{R}^3$ is the angular pre-collision velocity, $\boldsymbol{\omega}_i' \in \mathbb{R}^3$ is the angular post-collision velocity, $\boldsymbol{I}_i \in \mathbb{R}^{3\times3}$ is the inertia tensor in the world frame of reference, and $\boldsymbol{r}_i \in \mathbb{R}^3$ is offset of the contact point $\boldsymbol{p}$ from the CM.

The point $\boldsymbol{p}$ at the instant of collision is common to both bodies, denoted by $\boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{R}^3$ such that $\boldsymbol{p}_1 = \boldsymbol{p}_2 = \boldsymbol{p}$. The respective velocities $\boldsymbol{v}_{\boldsymbol{p}_1}, \boldsymbol{v}_{\boldsymbol{p}_2} \in \mathbb{R}^3$ of these points may be computed in terms of the respective linear and angular velocities, using

$$\boldsymbol{v}_{\boldsymbol{p}_i} = \boldsymbol{v}_i + \boldsymbol{\omega}_i \times \boldsymbol{r}_i \tag{2.62}$$

for $i = 1{,}2$. The coefficient of restitution e relates the pre-collision relative velocity $\boldsymbol{v}_\mathrm{r} = \boldsymbol{v}_{\boldsymbol{p}_2} - \boldsymbol{v}_{\boldsymbol{p}_1}$ of the contact point to the post-collision relative velocity $\boldsymbol{v}_\mathrm{r}' = \boldsymbol{v}_{\boldsymbol{p}_2}' - \boldsymbol{v}_{\boldsymbol{p}_1}'$ along the contact normal $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ via the coefficient of restitution e using

$$\boldsymbol{v}_\mathrm{r}' \cdot \hat{\boldsymbol{n}} = -e\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}} \tag{2.63}$$

Substituting equations (2.60a), (2.60b), (2.61a), (2.61b) and (2.62) into (2.63) and solving for the reaction impulse magnitude j_r yields

$$j_\mathrm{r} = \frac{-(1+e)\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}}}{\frac{1}{m_1} + \frac{1}{m_2} + \left(\boldsymbol{I}_1{}^{-1}(\boldsymbol{r}_1 \times \hat{\boldsymbol{n}}) \times \boldsymbol{r}_1 + \boldsymbol{I}_2{}^{-1}(\boldsymbol{r}_2 \times \hat{\boldsymbol{n}}) \times \boldsymbol{r}_2\right) \cdot \hat{\boldsymbol{n}}} \tag{2.64}$$

We observe that (2.64) remains consistent even when one of the masses is immovable, that is, possessing infinite mass. As $m_2, I_{2(ij)} \to \infty$, $\frac{1}{m_2} \to 0$ and $\boldsymbol{I}_2{}^{-1} \to \boldsymbol{0}$, resulting in an impulse equation involving the properties of only one body, such that

$$j_\mathrm{r} = \frac{-(1+e)\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}}}{\frac{1}{m_1} + \boldsymbol{I}_1{}^{-1}(\boldsymbol{r}_1 \times \hat{\boldsymbol{n}}) \times \boldsymbol{r}_1 \cdot \hat{\boldsymbol{n}}}$$

2.6.2.1 Impulse-based Friction

One of the most popular models for describing friction is the *Coulomb friction* model [**46**]. This model defines coefficients of static friction $\mu_\mathrm{s} \in \mathbb{R}$ and dynamic friction $\mu_\mathrm{d} \in \mathbb{R}$ such that $\mu_\mathrm{s} \geq \mu_\mathrm{d}$. These coefficients describe the two types of friction forces in terms of the reaction forces acting on the bodies. More specifically, the static and dynamic friction force magnitudes $f_\mathrm{s}, f_\mathrm{d} \in \mathbb{R}$ are computed in terms of the reaction force magnitude $f_\mathrm{r} = |\boldsymbol{f}_\mathrm{r}|$ as follows

$$f_\mathrm{s} = \mu_\mathrm{s} f_\mathrm{r} \tag{2.65a}$$

$$f_\mathrm{d} = \mu_\mathrm{d} f_\mathrm{r} \tag{2.65b}$$

The static friction magnitude f_s defines a maximum for the friction force required to counter the tangential component of any external sum force applied on a relatively static body, such that it remains static. Thus, if the external force is large enough, static friction is unable to fully counter this force, at which point the body gains velocity and becomes subject to dynamic friction of magnitude f_d acting against the sliding velocity.

The Coulomb friction model effectively defines a *friction cone* (Figure 32) within which a force exerted by one body on the surface of another in static contact, is countered by an equal and opposite force such that the static configuration is maintained. Conversely, if the force falls outside the cone, static friction gives way to dynamic friction.

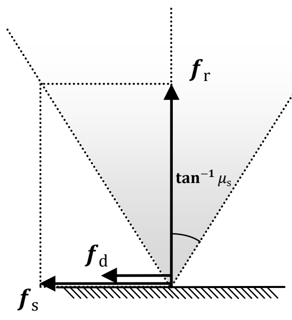

$\boldsymbol{f}_\mathrm{r}$
$\tan^{-1}\mu_\mathrm{s}$
$\boldsymbol{f}_\mathrm{d}$
$\boldsymbol{f}_\mathrm{s}$

Figure 32 Friction cone resulting from the Coulomb friction model

Given the contact normal $\hat{\boldsymbol{n}} \in \mathbb{R}^3$ and relative velocity $\boldsymbol{v}_\mathrm{r} \in \mathbb{R}^3$ of the contact point, a tangent vector $\hat{\boldsymbol{t}} \in \mathbb{R}^3$, orthogonal to $\hat{\boldsymbol{n}}$, may be defined such that

$$\hat{\boldsymbol{t}} = \begin{cases} \dfrac{\boldsymbol{v}_\mathrm{r} - (\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}}{|\boldsymbol{v}_\mathrm{r} - (\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}|} & \boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}} \neq 0 \\ \dfrac{\boldsymbol{f}_\mathrm{e} - (\boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}}{|\boldsymbol{f}_\mathrm{e} - (\boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}|} & \boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}} = 0 \quad \boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{n}} \neq 0 \\ \boldsymbol{0} & \boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{n}} = 0 \quad \boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{n}} = 0 \end{cases} \tag{2.66}$$

where $\boldsymbol{f}_\mathrm{e} \in \mathbb{R}^3$ is the sum of all external forces on the body. The multi-case definition of $\hat{\boldsymbol{t}}$ is required for robustly computing the actual friction force $\boldsymbol{f}_\mathrm{f} \in \mathbb{R}^3$ for both the general and particular states of contact. Thus, $\boldsymbol{f}_\mathrm{f}$ is computed as

$$\boldsymbol{f}_\mathrm{f} = \begin{cases} -(\boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{t}})\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} = \boldsymbol{0} \quad \boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{t}} \le f_\mathrm{s} \\ -f_\mathrm{s}\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} = \boldsymbol{0} \quad \boldsymbol{f}_\mathrm{e} \cdot \hat{\boldsymbol{t}} > f_\mathrm{s} \\ -f_\mathrm{d}\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} \ne \boldsymbol{0} \end{cases} \tag{2.67}$$

Equations (2.65a), (2.65b), (2.66) and (2.67) describe the Coulomb friction model in terms of forces, which is not amenable to the impulse-based reaction model described in (§ 2.6.2.1). By adapting the argument for instantaneous impulses, an impulse-based version of the Coulomb friction model may be derived, relating a frictional impulse $\boldsymbol{j}_\mathrm{f} \in \mathbb{R}^3$, acting along the tangent $\hat{\boldsymbol{t}}$, to the reaction impulse $\boldsymbol{j}_\mathrm{r}$. Integrating (2.65a) and (2.65b) over the collision time interval $[t_0 .. t_1]$ yields

$$j_\mathrm{s} = \mu_\mathrm{s} j_\mathrm{r} \tag{2.68a}$$

$$j_\mathrm{d} = \mu_\mathrm{d} j_\mathrm{r} \tag{2.68b}$$

where $j_\mathrm{r} = |\boldsymbol{j}_\mathrm{r}|$ is the magnitude of the reaction impulse acting along contact normal $\hat{\boldsymbol{n}}$. Similarly, by assuming $\hat{\boldsymbol{t}}$ constant throughout the time interval, the integration of (2.67) yields

$$\boldsymbol{j}_\mathrm{f} = \begin{cases} -(m\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{t}})\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} = \boldsymbol{0} \quad m\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{t}} \le j_\mathrm{s} \\ -j_\mathrm{s}\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} = \boldsymbol{0} \quad m\boldsymbol{v}_\mathrm{r} \cdot \hat{\boldsymbol{t}} > j_\mathrm{s} \\ -j_\mathrm{d}\hat{\boldsymbol{t}} & \boldsymbol{v}_\mathrm{r} \ne \boldsymbol{0} \end{cases} \tag{2.69}$$

Equations (2.64) and (2.69) define an impulse-based contact model that is ideal for impulse-based simulations [**16**]. When using this model, care must be taken in the choice of μ_s and μ_d as higher values may introduce additional kinetic energy into the system [**47**].

2.6.3 Penalty-based Contact Model

In the real world, the solidity of matter results from the strong repulsive forces at atomic level. A contact technique that attempts to simulate this phenomenon penalises body interpenetrations using a spring-damper system of forces [**48**]. The lifetime of these penalty forces is short-term, consisting of the time period in which the bodies are interpenetrating, and are thereafter removed from the simulation until new penetrations occur. However, the forces can be solved numerically, similarly to the other forces present in the simulation and hence require no additional contact handling sub-systems such as for impulse-based contact models.

2.6.3.1 Penalty-based Reaction

Ignoring the effects of friction, penalty-based reaction forces between two bodies are modelled by damped springs acting along the contact point normals, and connected to the respective contact features (Figure 33). The spring forces are set up such that deeper penetrations induce stronger tensile forces to repel the bodies away from each other.

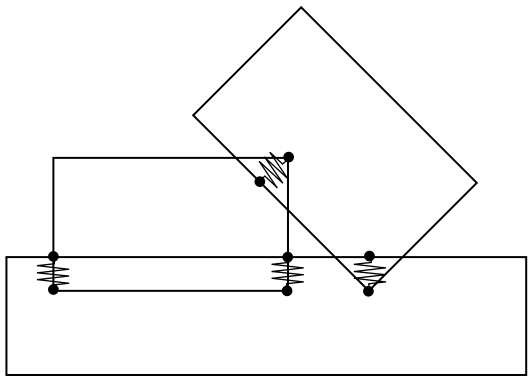


Figure 33 The application of damped springs to penalise body interpenetration

For a given contact point of the form $\boldsymbol{c} = (\boldsymbol{p}, \hat{\boldsymbol{n}}, d)$ between two bodies as defined in (2.58b), and assuming that contact position $\boldsymbol{p}$ in world space lies at the midpoint of the penetrating features of the respective bodies, such that their positions in world space are $\boldsymbol{p}_1 = \boldsymbol{p} - \frac{d}{2}\hat{\boldsymbol{n}}$ and $\boldsymbol{p}_2 = \boldsymbol{p} + \frac{d}{2}\hat{\boldsymbol{n}}$, the damped spring forces $\boldsymbol{f}_{\mathrm{r}1}, \boldsymbol{f}_{\mathrm{r}2} \in \mathbb{R}^3$ applied on $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ take the form

$$\boldsymbol{f}_{\mathrm{r}1} = k_s d\hat{\boldsymbol{n}} - k_d(\boldsymbol{v}_{\mathrm{r}} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} \quad (2.70a)$$

$$\boldsymbol{f}_{\mathrm{r}2} = -k_s d\hat{\boldsymbol{n}} + k_d(\boldsymbol{v}_{\mathrm{r}} \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} \quad (2.70b)$$

where $\boldsymbol{v}_{\mathrm{r}} \in \mathbb{R}^3$ is the relative velocity of the contact point and $k_s, k_d \in \mathbb{R}$ are appropriate spring and damping coefficients chosen to minimise penetration and oscillation between bodies in stacked configurations. The penalty forces $\boldsymbol{f}_{\mathrm{r}i}$ have a magnitude proportional to the contact penetration depth d, augmented by a damping term proportional to the magnitude $\boldsymbol{v}_{\mathrm{r}} \cdot \hat{\boldsymbol{n}}$ of the relative contact velocity along the contact normal.

A number of analytical and numerical techniques, such as [**49**] and [**50**], have been researched for deriving suitable values for k_s and k_d. The aim of these techniques is to yield coefficients that induce the penalty forces into *critically damped harmonic oscillation*. Although numerical techniques are more practical in real-time physics simulation, they are prone to resonance phenomena. The inverse dynamics approach [**51**] does not suffer from such phenomena. The technique entails setting up the desired critically damped oscillating motion and solving for the penalty forces required for this motion to occur.

2.6.3.2 Penalty-based Friction

A penalty-based form of the Coulomb friction model may also be constructed with associated static and dynamic friction coefficients μ_{s} and μ_{d}, as proposed by [**49**] and [**52**]. The concept entails storing an anchor point $\boldsymbol{a}_0 \in \mathbb{R}^3$ coincident with the contact point $\boldsymbol{p}$ on penetration.

Spring forces $\boldsymbol{f}_{\mathrm{s}1}, \boldsymbol{f}_{\mathrm{s}2} \in \mathbb{R}^3$ are then applied at $\boldsymbol{a}_0$ and the moving point $\boldsymbol{p}$ respectively, defined as

$$\boldsymbol{f}_{\mathrm{s}1} = k_s(\boldsymbol{p} - \boldsymbol{a}_0) \tag{2.71a}$$

$$\boldsymbol{f}_{\mathrm{s}2} = k_s(\boldsymbol{a}_0 - \boldsymbol{p}) \tag{2.71b}$$

The forces $\boldsymbol{f}_{\mathrm{s}1}$ and $\boldsymbol{f}_{\mathrm{s}2}$ are a crude representation of the real-world static friction forces within the microstructure of the surfaces in contact and are set up such that they resist tangential motion (Figure 34).

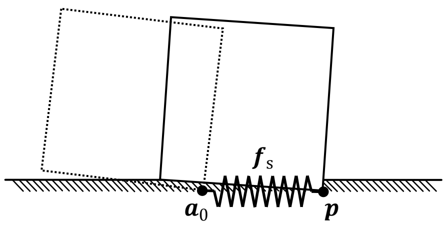


Figure 34 Static friction penalty force

The static friction forces are applied as long as the bodies are in contact and the following static friction condition, inspired from the Coulomb friction model, holds

$$|\boldsymbol{f}_{si}| \leq \mu_{\mathrm{s}} |\boldsymbol{f}_{\mathrm{r}i}| \tag{2.72}$$

for $i = 1,2$. If condition (2.72) is no longer satisfied, the friction model switches to dynamic friction by applying forces $\boldsymbol{f}_{\mathrm{d}1}, \boldsymbol{f}_{\mathrm{d}2} \in \mathbb{R}^3$, computed as

$$\boldsymbol{f}_{\mathrm{d}1} = k_s(\boldsymbol{p} - \boldsymbol{a}) \tag{2.73a}$$

$$\boldsymbol{f}_{\mathrm{d}2} = k_s(\boldsymbol{a} - \boldsymbol{p}) \tag{2.73b}$$

where the moving anchor point $\boldsymbol{a} \in \mathbb{R}^3$ is placed along the line joining $\boldsymbol{a}_0$ to $\boldsymbol{p}$ such that the following condition holds

$$|\boldsymbol{f}_{\mathrm{d}i}| = \mu_{\mathrm{d}} |\boldsymbol{f}_{\mathrm{r}i}|$$

for $i = 1,2$ (Figure 35). The moving anchor $\boldsymbol{a}$ need not be explicitly computed, and hence $\boldsymbol{f}_{\mathrm{d}1}$ and $\boldsymbol{f}_{\mathrm{d}2}$ may be computed more directly in terms of $\boldsymbol{a}_0$ using

$$\boldsymbol{f}_{\mathrm{d}1} = \mu_{\mathrm{d}} |\boldsymbol{f}_{\mathrm{r}1}| \frac{\boldsymbol{p} - \boldsymbol{a}_0}{|\boldsymbol{p} - \boldsymbol{a}_0|} \tag{2.74a}$$

$$\boldsymbol{f}_{\mathrm{d}2} = \mu_{\mathrm{d}} |\boldsymbol{f}_{\mathrm{r}2}| \frac{\boldsymbol{a}_0 - \boldsymbol{p}}{|\boldsymbol{a}_0 - \boldsymbol{p}|} \tag{2.74b}$$

provided that $\boldsymbol{p} \neq \boldsymbol{a}_0$. For $\boldsymbol{p} = \boldsymbol{a}_0$, the contact is static, in which case no friction applies.

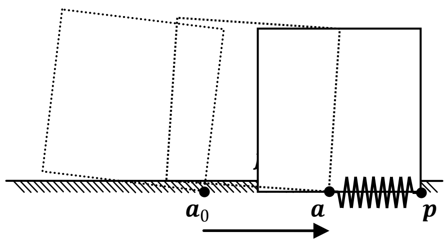


Figure 35 Dynamic friction penalty force

A variation of this contact model by [**53**] employs different spring coefficients for the tension and compression $k_\mathrm{t}, k_\mathrm{c} \in \mathbb{R}$ such that $k_\mathrm{t} = ek_\mathrm{c}$ where e is the coefficient of friction. In addition [**53**] implements a hybrid approach, treating colliding contacts using an algebraic method similar to the Newton collision law, while static or sliding contacts are handled using the penalty method.

2.6.4 Rolling and Spinning Friction

A contact model consisting of reaction and surface-to-surface friction solves many dynamics simulation problems. However, such a model is still a gross simplification of real contact phenomena, in particular, other forms of friction.

If a sphere in contact with an immovable ground plane (Figure 36), and under the influence of a constant field of gravity, is assigned a velocity tangential to the plane at its CM, the sphere acquires tangential momentum. Reaction forces prevent the sphere from falling through the plane, while friction forces resist its tangential motion at the point of contact, resulting in a resistive translational force at the CM that induces linear deceleration and a torque that induces angular acceleration. The relative velocity of the contact point eventually converges to zero, resulting in the sphere acquiring rolling motion such that both its linear and angular velocities remain constant for an indefinite period of time. Mathematically, rolling is achieved when the conditions $\boldsymbol{\omega} \neq \mathbf{0}$ and $\boldsymbol{v}_\mathrm{r} = \mathbf{0}$ hold true, where $\boldsymbol{\omega} \in \mathbb{R}^3$ is the body's relative angular velocity and $\boldsymbol{v}_\mathrm{r} \in \mathbb{R}^3$ is the relative velocity for some point of contact between the body and some other body.

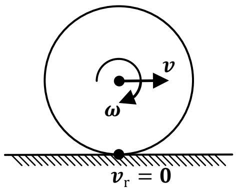


Figure 36 Body in a rolling configuration

The motion just described is correct for a perfect sphere rolling on a perfect ground plane, and hence a contact model defined in terms of reaction and friction is sufficient to describe its motion. In the real world however, a sphere rolling on level ground eventually slows down to a stop due to *rolling friction* that results from the microstructure of the contact surfaces and the microscopic compression of matter under the effect of forces, amongst other factors. A similar problem occurs when a stationary sphere in the same configuration as the previous example is assigned angular velocity normal to the ground plane. The sphere spins around the plane normal indefinitely with its contact point to the plane lying stationary.

A very simple model for rolling and spinning friction entails applying a friction coefficient $\mu_\mathrm{r} \in \mathbb{R}$ as a damping term to the angular velocity $\boldsymbol{\omega} \in \mathbb{R}^3$, such that $\boldsymbol{\omega}(t + \Delta t) = \boldsymbol{\omega}(t)e^{-\mu_\mathrm{r}\Delta t}$, ignoring other factors influencing $\boldsymbol{\omega}$. This differential equation may be incorporated into the numerical integration for $\boldsymbol{\omega}$ by adding the derivative terms of the Taylor series $\Delta t\dot{\boldsymbol{\omega}}(t)$, $\frac{\Delta t^2}{2}\ddot{\boldsymbol{\omega}}(t)$, $\frac{\Delta t^3}{6}\dddot{\boldsymbol{\omega}}(t)$ *etc.* as appropriate, depending on the numerical integration method used. The reduction in angular velocity breaks the condition $\boldsymbol{v}_\mathrm{r} = \mathbf{0}$ in the case of rolling motion and the resulting surface-to-surface friction force decreases the linear velocity $\boldsymbol{v} \in \mathbb{R}^3$ of the body until it converges to zero. Similar results for rolling motion may be achieved by applying a linear damping coefficient to $\boldsymbol{v}$ analogous to a first order fluid drag model. The drawback of these techniques however, is that they cause indiscriminate angular damping even when the body is not in a rolling or spinning configuration.

An improvement over this technique attempts to detect the rolling or spinning state of a body by verifying the rolling conditions $\boldsymbol{\omega} \neq \mathbf{0}$ and $\boldsymbol{v}_\mathrm{r} = \mathbf{0}$ for some contact point. If the conditions hold true, a damping coefficient or an appropriate resistive force is applied. These conditions generally indicate a combination of spinning and rolling motion around a contact point. The contribution of each type of motion at the contact point may be determined by projecting the angular velocity onto the contact normal and appropriate rolling and spinning friction forces may be applied.

A different approach entails constructing contact manifolds such that the reaction forces include a tangential component. For a sphere, this may entail building a manifold of two contact points: one at the usual point of contact; and another ahead of this contact along the direction of relative velocity (Figure 37), and such that its normal points towards the sphere centre. The second contact results in a reaction force with a component oriented such that it resists tangential motion.

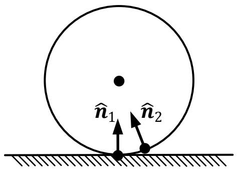


Figure 37 Extended contact manifolds to induce rolling friction

A related approach to building contact manifolds with specially oriented normals entails using polyhedron-based geometry instead of primitive-based geometry. Thus, rolling and spinning friction is achieved as a by-product of the multi-point contact manifolds generated for polyhedral geometry. For instance, a sphere or a cylinder may be replaced by polyhedral approximations of sufficient detail (Figure 38). The drawback of this approach is the increased computational cost of collision detection for such geometry.

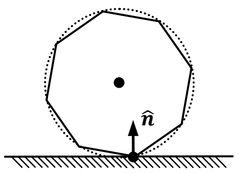


Figure 38 Using polyhedral geometry approximations to induce rolling friction

2.6.5 Surface Imperfections

Idealised geometry produces mathematically correct but, in a number of contrived situations, implausible motion (§ 3.1.4). In real physical experiments such situations do not occur due to microscopically imperfect surfaces (Figure 39) that introduce an element of chaos in the predicted motion.

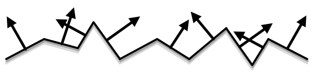


Figure 39 Varying contact normals due to surface microstructure imperfections

Attempts at modelling these surface microstructures using detailed geometry result in intractable computations due to the sheer complexity of such geometry. Hence, simpler localised models are applied at the point of contact to produce slightly perturbed normals. One technique entails defining a smoothed noise function $f\colon \mathbb{R}^3 \longmapsto \boldsymbol{N}$ where $\boldsymbol{N} = \{\hat{\boldsymbol{n}} \in \mathbb{R}^3 | |\hat{\boldsymbol{n}}| = 1\}$ is the set of unit vectors, geometrically equivalent to the unit sphere. The point of contact $\boldsymbol{p} \in \mathbb{R}^3$ may hence be fed to f to yield a contact normal $\hat{\boldsymbol{n}}$ associated with $\boldsymbol{p}$.

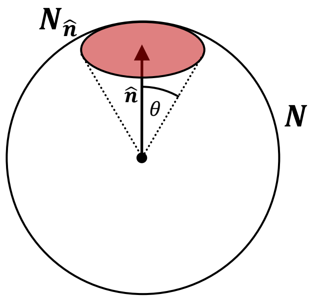


Figure 40 Perturbing a surface normal within a spherical cap

Due to the chaotic nature of simulations, the determinate function f may be replaced by a random variable $\hat{\boldsymbol{n}}_{\mu} \in \boldsymbol{N}_{\hat{\boldsymbol{n}}} \subset \boldsymbol{N}$ where $\boldsymbol{N}_{\hat{\boldsymbol{n}}} = \{\hat{\boldsymbol{m}} \in \boldsymbol{N} | \hat{\boldsymbol{m}} \cdot \hat{\boldsymbol{n}} > 1 - \sigma\}$ effectively defines a *spherical cap* $\boldsymbol{N}_{\hat{\boldsymbol{n}}}$ of the unit sphere set $\boldsymbol{N}$, subtended by a *steradian* angle of $2\theta =$

$2\cos^{-1}(1-\sigma)$ and centred around the ideal surface normal $\hat{\boldsymbol{n}}$ (Figure 40). The coefficient $0 \leq \sigma \leq 1$ is essentially a variance parameter defining the level of surface perturbation.

Ignoring the exact nature of the probability distribution function over $\boldsymbol{N}_{\hat{\boldsymbol{n}}}$, the random variable $\hat{\boldsymbol{n}}_\mu$ may be approximated by simply perturbing the coefficients of idealised surface normal $\hat{\boldsymbol{n}}$ by pseudorandom variables $x_\mu, y_\mu, z_\mu \in \mathbb{R}$ selected uniformly from the range $[-\varepsilon \ldots \varepsilon]$. The resulting vector is then normalised to yield $\hat{\boldsymbol{n}}_\mu$, as follows

$$\hat{\boldsymbol{n}}_\mu = \frac{\hat{\boldsymbol{n}} + [x_\mu \quad y_\mu \quad z_\mu]}{|\hat{\boldsymbol{n}} + [x_\mu \quad y_\mu \quad z_\mu]|}$$

where $0 \leq \varepsilon \leq 1$ is a surface perturbation term. For plausible results, we recommend relatively small values for ε, typically $0.05 \leq \varepsilon \leq 0.1$.

2.6.6 Local Collision Resolution

Collision resolution entails solving velocities, and where applicable, the interpenetration of bodies in collision or in contact. One of the simplest contact resolution algorithms entails iterating through each contact manifold in the current simulation step, selecting and resolving the deepest contact within the manifold using equations such as (2.64) and (2.69). A variation of this algorithm sorts all contact points in order of decreasing depth, regardless of the manifold in which they are contained and iteratively solves all the contacts in that order. Such algorithms are based on the assumption that resolving deeper contacts before shallower ones increases the likelihood of convergence to a global solution where all contact conditions are satisfied.

Contact penetration is solved locally using a number of techniques. The simplest penetration resolution technique between two bodies of mass $m_1 \in \mathbb{R}$ and $m_2 \in \mathbb{R}$ at a single contact point with penetration depth d, along contact normal $\hat{\boldsymbol{n}}$, is a mass-weighted linear displacement (Figure 41) of the bodies' positions $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^3$ to new positions $\boldsymbol{x}_1', \boldsymbol{x}_2' \in \mathbb{R}^3$ computed as

$$\boldsymbol{x}_1' = \boldsymbol{x}_1 - \frac{m_2 d}{m_1 + m_2}\hat{\boldsymbol{n}} \tag{2.75a}$$

$$\boldsymbol{x}_2' = \boldsymbol{x}_2 + \frac{m_1 d}{m_1 + m_2}\hat{\boldsymbol{n}} \tag{2.75b}$$

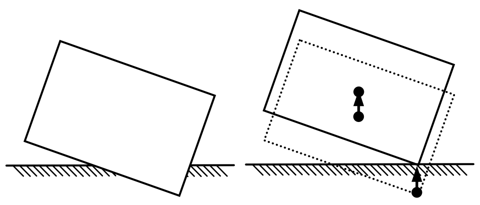


Figure 41 Solving interpenetration by linear translation

For simple configurations, these algorithms have a small computational footprint, are simple to implement and generate adequate contact behaviour. However, for configurations involving stacked bodies, these algorithms generate a train of impulses up and down the stack across a number of simulation steps, resulting in vibrating bodies and eventual collapse of the stack. Additionally, local penetration resolution solves penetration between two given bodies, but may induce penetrations elsewhere.

2.6.7 Simultaneous Collision Resolution

The collision resolution techniques discussed in (§§ 2.6.2, 2.6.3, 2.6.6) treat the contact problem locally without taking into consideration the global problem. A contact manifold may contain multiple points of contact and in addition, a body may be in simultaneous contact with several bodies. Hence, attempting to resolve one particular contact may induce further interpenetration at other points of contact and potentially create new contact points between the two bodies or other neighbouring bodies (Figure 42).

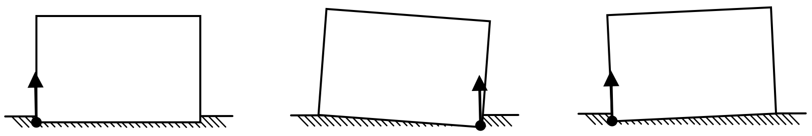


Figure 42 Limitations of a local collision resolver

To counter these issues, researchers have developed a number of advanced algorithms that attempt to simultaneously resolve all contacts within a single simulation step. Simultaneous methods fall under two main categories: iterative methods and linear complementary problem (LCP) formulations.

Penalty-based methods for collision resolution do not require an explicit collision resolver, as the force integration sub-system inherently handles the contact forces. However, these methods are mentioned in passing as they are inherently simultaneous resolution methods.

2.6.7.1 Iterative Methods

Iterative methods, based on the *relaxation method*, are arguably the simpler of the two approaches and generally operate by sequentially solving a fraction of each contact problem locally over several iterations within a single simulation step. Relaxation allows the algorithm to converge to an optimal or near optimal solution that satisfies all the post-contact velocity and non-penetration conditions.

One of the simpler iterative algorithms initially computes values for the required local contact impulses and position projections using equations such as (2.64), (2.69), (2.75a) and (2.75b). To illustrate the concept, we define $\boldsymbol{p}_{ij}, \boldsymbol{j}_{ij}, \Delta\boldsymbol{x}_{ij} \in \mathbb{R}^3$ respectively as the contact position, combined contact impulse and displacement vectors for solving interpenetration at the ith contact of the jth body. The contact indices $i \in \mathbb{N}$ range over $1 \leq i \leq n_j$, where $n_j \in \mathbb{N}$ is the number of contacts involving the jth body. The algorithm iterates for a number of times $k \in \mathbb{N}$ over the n_j contacts of each body indexed by j, computing new values for the linear and angular velocities and centroid positions as follows

$$\boldsymbol{v}_j[l+1] = \boldsymbol{v}_j[l] + \frac{1}{km_l}\sum_{i=1}^{n_j}\boldsymbol{j}_{ij} \tag{2.76a}$$

$$\boldsymbol{\omega}_j[l+1] = \boldsymbol{\omega}_j[l] + \frac{1}{k}\boldsymbol{I}_j^{-1}\sum_{i=1}^{n_j}\boldsymbol{r}_{ij} \times \hat{\boldsymbol{n}}_{ij} \tag{2.76b}$$

$$\boldsymbol{x}_j[l+1] = \boldsymbol{x}_j[l] + \frac{1}{k}\sum_{i=1}^{n_j}\Delta\boldsymbol{x}_{ij} \tag{2.76c}$$

where $1 \leq l \leq k$ is the lth iteration, $\boldsymbol{v}_j[0], \boldsymbol{\omega}_j[0], \boldsymbol{x}_j[0] \in \mathbb{R}^3$ are the pre-collision velocities and position of the jth body, and $\boldsymbol{v}_j[l], \boldsymbol{\omega}_j[l], \boldsymbol{x}_j[l]$ are their respective update values at the lth iteration.

The advantages of iterative algorithms are that they are generally faster than LCP-based formulations, and accuracy may be traded with performance by varying the number of iterations k. For $k = 1$, the algorithm effectively degenerates to a local collision resolver.

2.6.7.2 Linear Complementary Problem-based Formulations

LCP-based formulations such as [**54**] handle static and sliding contact cases by setting up non-penetration conditions for n contact points in the unified matrix and vector form

$$\boldsymbol{a} = \boldsymbol{A}\boldsymbol{f} + \boldsymbol{b} \geq \boldsymbol{0} \tag{2.77a}$$

$$\boldsymbol{f} \cdot \boldsymbol{a} \geq \boldsymbol{0} \tag{2.77b}$$

$$\boldsymbol{f} \geq \boldsymbol{0} \tag{2.77c}$$

$$\boldsymbol{a} \geq \boldsymbol{0} \tag{2.77d}$$

Each of the first n coefficients $a_1 .. a_n \in \mathbb{R}$ of $\boldsymbol{a} \in \mathbb{R}^{2n}$ represents the relative acceleration component of the ith contact point along the contact normals $\hat{\boldsymbol{n}}_i$. Similarly, each of the remaining coefficients $a_{n+1} .. a_{2n} \in \mathbb{R}$ of $\boldsymbol{a}$ represents the relative acceleration component of the ith contact point along the contact tangents $\hat{\boldsymbol{t}}_i$ such that $\hat{\boldsymbol{t}}_i$ is in the direction of friction, where applicable. Likewise, the coefficients $f_1 .. f_n \in \mathbb{R}$ and $f_{n+1} .. f_{2n} \in \mathbb{R}$ of the $2n$-dimensional vector $\boldsymbol{f} \in \mathbb{R}^{2n}$ represent the required reaction and static friction forces respectively along the contact normals $\hat{\boldsymbol{n}}_i$ and tangents $\hat{\boldsymbol{t}}_i$, such that interpenetration does not occur.

The complementary condition (2.77b) may be expressed in its element-wise form

$$f_i > 0 \Leftrightarrow a_i = 0$$

$$f_i = 0 \Leftrightarrow a_i > 0$$

for $1 \leq i \leq 2n$. A positive value for a_i signifies a relative acceleration along $\hat{\boldsymbol{n}}_i$ of a separating nature. Hence, no contact force is required and hence $f_i = 0$. Conversely, $a_i = 0$ implies relatively static normal or tangential velocity, and hence a reaction or friction force $f_i > 0$ is required to maintain the sticking or sliding contact.

The relative acceleration coefficients of $\boldsymbol{a}$ are expressed as a vector of linear functions $\boldsymbol{a} = \boldsymbol{A}\boldsymbol{f} + \boldsymbol{b}$ where the elements of the positive semi-definite (PSD) matrix $\boldsymbol{A} \in \mathbb{R}^{2n \times 2n}$ are dependent on the mass properties and contact placement of the bodies in the system, while the coefficients of $\boldsymbol{b} \in \mathbb{R}^{2n}$ represent the system's inertial and external forces.

The goal of the LCP formulation is to compute a value for $\boldsymbol{f}$ such that the linear complementary conditions hold true. The formulation may also be adapted to include dynamic friction by replacing the appropriate elements in $\boldsymbol{A}$. This may however violate the PSD property of $\boldsymbol{A}$, making the problem unsolvable as a LCP. However, by using Lemke's algorithm [**55**] to solve the LCP, termination still occurs and the last computed value for $\boldsymbol{f}$ may still be used. The required reaction and friction forces $\boldsymbol{f}_{\mathrm{r}i}, \boldsymbol{f}_{\mathrm{f}i}$ are computed from f_i as

$$\boldsymbol{f}_{\mathrm{r}i} = f_i \hat{\boldsymbol{n}}_i$$

$$\boldsymbol{f}_{\mathrm{f}i} = f_{n+i} \hat{\boldsymbol{t}}_i$$

for $1 \leq i \leq n$, and are applied at the corresponding contact positions $\boldsymbol{p}_i \in \mathbb{R}^3$ of the respective bodies.

LCP formulations solved using the Lemke method are advantageous because of their low polynomial time complexity [**56**] and because they fit more intuitively with analytical approaches that set up motion equations and solve for the required forces. By considering the first order form of Newtonian equations of motion, LCP techniques may also be used to solve interpenetration using a simultaneous approach.

2.6.7.3 Block Solvers

Due to the polynomial time complexity of simultaneous collision resolvers, simulations involving larger amounts of bodies incur a substantially higher computational expense, limiting the complexity of simulations running in real-time. This problem is mitigated by partitioning the complete set of bodies comprising the simulation, represented by the set $\boldsymbol{U} = \{1,2,\ldots,n\} \subset \mathbb{N}$ of body indices, into a number of disjoint subsets $\boldsymbol{S}_1, \boldsymbol{S}_2 .. \boldsymbol{S}_m$, of bodies in direct or indirect contact (Figure 43), termed *islands* by [**22**], such that a simultaneous solver may be independently applied to each subset.

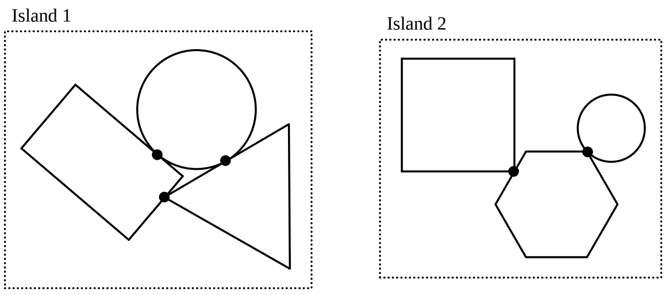


Figure 43 Partitioning a simulation into islands of bodies in contact

The elements of each subset $\boldsymbol{S}_i$ index the bodies that are directly or indirectly in contact with each other. A direct contact predicate function $\boldsymbol{p}_\mathrm{c}: \boldsymbol{U} \times \boldsymbol{U} \longmapsto \mathbb{B}$ may be defined by

$$\boldsymbol{p}_\mathrm{c}(i,j) \overset{\mathrm{def}}{=} \chi_i \cap \chi_i \neq \emptyset \tag{2.78}$$

where χ_i is the region of the ith body geometry in world space. An indirect contact predicate function $\boldsymbol{p}_\mathrm{d}: \boldsymbol{U} \times \boldsymbol{U} \longmapsto \mathbb{B}$ may be defined recursively in terms of the direct contact predicate $\boldsymbol{p}_\mathrm{c}$ by

$$\boldsymbol{p}_\mathrm{d}(i,j) \overset{\mathrm{def}}{=} \begin{cases} T & \boldsymbol{p}_\mathrm{c}(i,j) \\ \boldsymbol{p}_\mathrm{c}(i,k) \wedge \boldsymbol{p}_\mathrm{d}(k,j) & \sim\boldsymbol{p}_\mathrm{c}(i,j) \quad i \neq k \neq j \end{cases} \tag{2.79}$$

The first subset $\boldsymbol{S}_1$ may be constructed by picking an index from $\boldsymbol{U}$ and inserting all other indices from the remaining elements of $\boldsymbol{U}$ such that the indirect contact predicate $\boldsymbol{p}_\mathrm{d}$ holds true. Formally, $\boldsymbol{S}_1$ may be defined as

$$\boldsymbol{S}_1 = \{1 \in \boldsymbol{U}\} \cup \{j \in \boldsymbol{U}\backslash\{1\} | \boldsymbol{p}_\mathrm{d}(1,j)\}$$

A similar construction applies to subset $\boldsymbol{S}_2$ by picking a new index from $\boldsymbol{U}\backslash\boldsymbol{S}_1$ and inserting all other indices from this set where the $\boldsymbol{p}_\mathrm{d}$ holds true. The definition for $\boldsymbol{S}_2$ is

$$\boldsymbol{S}_2 = \{i \in \boldsymbol{U}\backslash\boldsymbol{S}_1\} \cup \{j \in \boldsymbol{U}\backslash(\{i\} \cup \boldsymbol{S}_1) | \boldsymbol{p}_\mathrm{d}(i,j)\}$$

The general construction of the kth partition $\boldsymbol{S}_k$ is thus

$$\boldsymbol{S}_k = \{i \in \boldsymbol{U}\backslash\boldsymbol{V}_{k-1}\} \cup \{j \in \boldsymbol{U}\backslash(\{i\} \cup \boldsymbol{V}_{k-1}) | \boldsymbol{p}_\mathrm{d}(i,j)\} \tag{2.80}$$

where $\boldsymbol{V}_k = \boldsymbol{S}_1 \cup \boldsymbol{S}_2 \ldots \cup \boldsymbol{S}_k$. This construction approach may be proven to terminate and yield a finite number m of partitions $\boldsymbol{S}_i$ spanning $\boldsymbol{U}$ for some $m \in \mathbb{N}$.

The first part of the proof verifies termination of the construction technique as follows: Every subset $\boldsymbol{S}_k$ is in the form $\boldsymbol{S}_k = \boldsymbol{A}_k \cup \boldsymbol{B}_k$ where

$$\boldsymbol{A}_k = \{i \in \boldsymbol{U}\backslash\boldsymbol{V}_{k-1}\}$$

$$\boldsymbol{B}_k = \{j \in \boldsymbol{U}\backslash(\{i\} \cup \boldsymbol{V}_{k-1}) | \boldsymbol{p}_\mathrm{d}(i,j)\}$$

If $\boldsymbol{A}_k = \emptyset$, then by definition $\boldsymbol{B}_k = \emptyset$ and $\boldsymbol{U} = \boldsymbol{V}_{k-1}$. Otherwise, $|\boldsymbol{A}_k| = 1$ and the subset $\boldsymbol{B}_k$ is a set of all elements that are in direct or indirect contact with the only element of $\boldsymbol{A}_k$. No specific conclusion may be drawn about the size of $\boldsymbol{B}_k$, other than $|\boldsymbol{B}_k| \geq 0$. However by definition, $\boldsymbol{A}_k \cap \boldsymbol{B}_k = \emptyset$, giving $|\boldsymbol{A}_k \cap \boldsymbol{B}_k| = 0$ and hence, $|\boldsymbol{S}_k| = |\boldsymbol{A}_k| + |\boldsymbol{B}_k| - |\boldsymbol{A}_k \cap \boldsymbol{B}_k| = |\boldsymbol{A}_k| + |\boldsymbol{B}_k| \geq 1$. Thus, $\boldsymbol{S}_k \neq \emptyset$ for all constructible subsets $\boldsymbol{S}_k$.

The disjointedness of the sets $\boldsymbol{S}_i$ may be proven by contradiction using the hypothesis $\boldsymbol{S}_i \cap \boldsymbol{S}_j \neq \emptyset$ for $i < j$, then for some $k \in \boldsymbol{U}$, $k \in \boldsymbol{S}_i \cap \boldsymbol{S}_j$, that is $k \in \boldsymbol{S}_i$ and $k \in \boldsymbol{S}_j$. By definition of the subsets $\boldsymbol{S}_i$,

$$\boldsymbol{S}_i \subseteq \boldsymbol{U} \backslash \boldsymbol{V}_{i-1}$$

$$\boldsymbol{S}_j \subseteq \boldsymbol{U} \backslash \boldsymbol{V}_{j-1}$$

and hence

$$k \notin \boldsymbol{V}_{i-1} = \boldsymbol{S}_1 \cup \boldsymbol{S}_2 \ldots \boldsymbol{S}_{i-1}$$

$$k \notin \boldsymbol{V}_{j-1} = \boldsymbol{S}_1 \cup \boldsymbol{S}_2 \ldots \boldsymbol{S}_i \ldots \boldsymbol{S}_{j-1}$$

that is, $k \notin \boldsymbol{S}_l$ for all $1 \leq l \leq j$. In particular,

$$k \notin \boldsymbol{S}_i, k \notin \boldsymbol{S}_j \Rightarrow k \notin \boldsymbol{S}_i \cap \boldsymbol{S}_j$$

But this contradicts the statement $k \in \boldsymbol{S}_i \cap \boldsymbol{S}_j$ that also follows from the same hypothesis and therefore $\boldsymbol{S}_i \cap \boldsymbol{S}_j = \emptyset$ for $i < j$. Without loss of generality, i and j may be interchanged, giving the proof $\boldsymbol{S}_i \cap \boldsymbol{S}_j = \emptyset$ for $i > j$ and hence $\boldsymbol{S}_i \cap \boldsymbol{S}_j = \emptyset$ for $i \neq j$.

The disjointedness property and the guaranteed termination of the construction process for $\boldsymbol{S}_i$ implies that $d = |\boldsymbol{U} \backslash \boldsymbol{V}_{i-1}| = |\boldsymbol{U}| - |\boldsymbol{V}_{i-1}| = |\boldsymbol{U}| - |\boldsymbol{S}_1| - \cdots - |\boldsymbol{S}_{i-1}|$ is a monotonically decreasing discrete function of i, such that for some $m \in \mathbb{N}$, d reaches zero, resulting in $m \leq |\boldsymbol{U}|$ disjoint subsets $\boldsymbol{S}_i$ spanning $\boldsymbol{U}$. This completes the proof that subsets $\boldsymbol{S}_1, \boldsymbol{S}_2 .. \boldsymbol{S}_m$ partition $\boldsymbol{U}$ into distinct islands of bodies in direct or indirect contact.

In the worst case, $m = 1$ and $\boldsymbol{U} = \boldsymbol{S}_1$ implies a configuration where all bodies are in direct or indirect contact and hence must be solved as one complete system of contacts. The best case occurs when $m = |\boldsymbol{U}|$ and $|\boldsymbol{S}_i| = 1$ for all partitions, indicating a configuration where all bodies are detached from each other, and no collision resolution is required.

2.7 Constraints

Articulated joints, rigid joints, cables, and other such constraints applied on bodies in a physics engine simulation are mathematically represented by unilateral inequalities or bilateral equations imposed on the kinetic properties of the affected bodies. A constraint between two bodies effectively reduces the number of degrees of freedom in the movement of one body with respect to the other.

2.7.1 Lagrangian-based Dynamics

For grounded or ungrounded tree-like articulated body configurations with no cyclic connections (Figure 44), the Featherstone method [**57**] and its more recent adaptations [**58**], solve the motion in terms of the degrees of freedom allowed by the joints of the articulated body. This method and other related techniques are based on Lagrangian formulations of mechanics where the constraints are implicitly defined in the equations of motion. These formulations offer advantages in terms of reduced computational complexity and accuracy. However, they are only applicable to specific types of constrained body motion, and hence we do not consider them further in this paper.

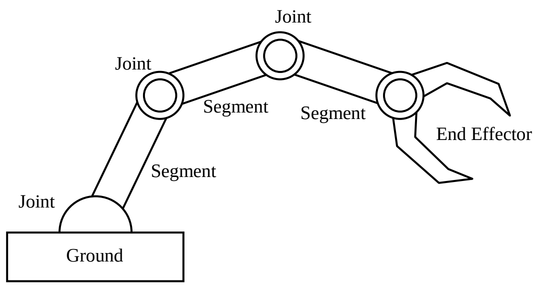


Figure 44 Articulated body amenable to the Featherstone method

2.7.2 Newtonian Dynamics

More general techniques integrate the simulation forward in time, allowing for temporary invalidation of the constraints, usually termed *joint errors*. A constraint solver algorithm is then applied to re-enforce the constraints, typically by applying appropriate forces, impulses or instantaneous changes in placement of the bodies to reduce or eliminate the joint errors. This is analogous to the collision resolution techniques discussed in (§§ 2.6.6, 2.6.7) where the penetration and velocity of the contacts are resolved to enforce the constraints dictated by the contact model.

2.7.3 Unified Constraint Models

The similarity between contacts, joints and other such constraints suggests a unified approach for collision and constraint resolution, and this is indeed the standard approach adopted by physics engine implementations, such as [**22**], [**23**], [**24**], [**59**], [**60**] and [**61**]. Thus, a contact point may be converted to a unilateral constraint defining the desired change in position to counter interpenetration and post-collision relative velocity as the contact equivalent of joint error.

The predominant unified representation for joint and contact constraints is the *Jacobian matrix* [**62**], often termed simply as Jacobian, representing a transform of the kinetic properties from the Cartesian space to the constraint space of the involved bodies. Jacobian formulations may be used as an abstraction for several types of constraints, as exemplified in the object-oriented approach by [**63**]. The advantage of this formulation is that collision, contact, joint and motor constraints may all be represented by appropriate Jacobians that can be fed to different types of sequential or simultaneous solvers [**64**]. The disadvantage is that the Jacobian formulation is tied to a specific set of approaches for constraint resolution, and in this paper we seek a more abstract representation that is agnostic of the underlying techniques. Nevertheless, we provide a cursory overview for the construction and application of Jacobian-based constraints as a further means for justifying a more abstract representation.

2.7.3.1 Jacobian Matrix Formulation

The derivation of a Jacobian for a bilateral constraint originates from a positional constraint equation in the form

$$c(\boldsymbol{x}) = 0$$

where c is a function $c: \mathbb{R}^n \longmapsto \mathbb{R}$ of the unified positional vector $\boldsymbol{x} \in \mathbb{R}^n$. The dimension n of $\boldsymbol{x}$ may be expressed as $n = kd$ where $k, d \in \mathbb{N}$ are respectively the number of bodies involved in the constraint and the number of degrees of freedom of a single unconstrained body. For a constraint involving two rigid bodies, $\boldsymbol{x}$ represents the position and orientation of both bodies. Given a minimal formulation for orientations $\boldsymbol{\theta}_i \in \mathbb{R}^3$ such that $\dot{\boldsymbol{\theta}}_i = \boldsymbol{\omega}_i$ where $\boldsymbol{\omega}_i$ are the angular velocities, $\boldsymbol{x} = [\boldsymbol{x}_1 \quad \boldsymbol{\omega}_1 \quad \boldsymbol{x}_2 \quad \boldsymbol{\omega}_2] \in \mathbb{R}^{12}$, that is, $n = 2 \times 6 = 12$. For a constraint involving a single point mass, $= [x_1 \quad x_2 \quad x_3] \in \mathbb{R}^3$, such that $n = 3$, is a sufficient representation. The Jacobian $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ is a matrix linearly relating the time derivative $\dot{c}$ of c to the time derivative $\boldsymbol{v} \in \mathbb{R}^n$ of $\boldsymbol{x}$ such that

$$\dot{c} = \boldsymbol{J}\boldsymbol{v}$$

Effectively $\boldsymbol{J}$ is a singular matrix, transforming the relative constraint velocity from Cartesian space to constraint space. The number of rows m of $\boldsymbol{J}$ correspond to the number of degrees of freedom removed by the constraint. A Jacobian may be derived by computing $\dot{c}$, isolating $\boldsymbol{v}$ and identifying $\boldsymbol{J}$ by inspection.

A constraint also determines the relative velocities of the involved bodies with respect to each other. For a time-independent constraint such as a ball-and-socket joint that anchors two bodies at a specific point common to both, the relative velocity is constrained to zero. In this case, the constraint derivative $\dot{c}$ obeys the equality

$$\dot{c} = \boldsymbol{J}\boldsymbol{v} = 0$$

For time-dependent constraints, such as angular motors that prescribe relative motion at the anchor axes, the positional constraint is defined by

$$c(\boldsymbol{x}, t) = 0$$

where c is parameterised by both $\boldsymbol{x}$ and time $t \in \mathbb{R}$. The time-dependent constraint velocity $\dot{c}$ is augmented by velocity bias term $b(t) \in \mathbb{R}$ to yield

$$\dot{c}(t) \ = \boldsymbol{J}\boldsymbol{v} + b(t) = 0$$

For unilateral constraints such as non-penetrative contact and collision, the positional constraint is formulated as the inequality

$$c(\boldsymbol{x}, t) \geq 0$$

In this case, the corresponding velocity constraint inequality $\dot{c}(t) \geq 0$ is enforced only when $c(\boldsymbol{x}, t) \leq 0$. By considering body contact as an example, the non-penetration velocity constraint is enforced only when there is penetration or exact contact.

2.7.3.2 Jacobian Constraint Solvers

The constraint forces required to maintain validity of the constraints, given in unified matrix form $\boldsymbol{F} \in \mathbb{R}^{m \times n}$, are related to the Jacobian by

$$\boldsymbol{F} = \lambda \boldsymbol{J}$$

where $\lambda \in \mathbb{R}$, termed the *Lagrange multiplier*, is the constraint force signed magnitude. Thus, the goal of a Jacobian constraint solver is to compute λ from which the constraint forces $\boldsymbol{F}$ may hence be derived and applied in force or impulse form to maintain the velocity constraints. The Lagrange multiplier may be computed by solving, for λ, the motion, virtual work and velocity constraint equations

$$\boldsymbol{v}' = \boldsymbol{v} + \boldsymbol{M}^{-1}\boldsymbol{P}$$

$$\boldsymbol{P} = \boldsymbol{J}^{\mathrm{T}}\lambda$$

$$\boldsymbol{J}\boldsymbol{v}' + b = 0$$

where $\boldsymbol{v}' \in \mathbb{R}^{n}$ is the corrected velocity, $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ is the mass matrix and $\boldsymbol{P} \in \mathbb{R}^{n \times m}$ is the first-order change in momentum $\boldsymbol{P} = \Delta t \boldsymbol{F}$ where $\Delta t \in \mathbb{R}$ is the simulation time-step. The matrix $\boldsymbol{M}$ unifies the linear and angular mass properties of k bodies through the formulation

$$\boldsymbol{M} = \begin{bmatrix} \begin{matrix} m_1\mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I}_1 \end{matrix} & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \begin{matrix} m_k\mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I}_k \end{matrix} \end{bmatrix}$$

where $m_i \in \mathbb{R}$ and $\boldsymbol{I}_i \in \mathbb{R}^{3 \times 3}$ are respectively the mass and inertia tensor of the ith body and $\mathbf{1}$ is the three-by-three identity matrix. In this representation, each body is expressed in six degrees of freedom, that is, $\boldsymbol{M} \in \mathbb{R}^{n \times n} = \mathbb{R}^{6k \times 6k}$.

Given the constraint forces $\boldsymbol{F}$ or impulses $\boldsymbol{P}$, a global or iterative solver may be used to enforce the constraint velocities. Iterative solvers are generally preferred for real-time simulations due to their speed. An iterative impulse-based solver applies the impulses $\boldsymbol{P}$ over several iterations, each time recomputing Jacobians and hence new values for $\boldsymbol{P}$, resulting in convergence to a global solution as each iteration yields subsequently smaller joint errors. The termination criteria for an iterative solver may include a limit on the number of iterations, a joint error tolerance, or a mixture of several such criteria.

Assuming computation at unlimited precision and a solver capable of computing optimal solutions, the application of constraint forces or impulses is sufficient to maintain the positional constraints. In practice, limited digital precision invariably causes positional drift over time. This is usually mitigated using Baumgarte Stabilization [**65**] that entails feeding the positional error back to the velocity constraint, such that

$$\dot{c} + \frac{\beta}{\Delta t} c = 0$$

where $0 \leq \beta \leq 1$ is a bias factor. This factor controls the rate of positional correction and is usually tuned for specific simulation configurations to avoid instability.

2.7.4 An Abstract Unified Constraint Model

In this section we propose what we hope is a novel, or at least an alternative formulation for describing a wide variety of constraints, that is abstracted from the underlying resolution techniques.

We define a constraint between two given bodies as a vector $\boldsymbol{c} \in \mathbb{R}^{10m}$ of m *constraint discrepancies*

$$\boldsymbol{c} = [\boldsymbol{d}_1 \quad \boldsymbol{d}_2 \quad \dots \quad \boldsymbol{d}_m] \tag{2.81a}$$

where each discrepancy $\boldsymbol{d}_i \in \mathbb{R}^{10}$ is defined by the quadruple

$$\boldsymbol{d}_i = (\boldsymbol{x}_{1i}, \boldsymbol{x}_{2i}, \boldsymbol{v}_{\mathrm{r}i}, \beta_i) \tag{2.81b}$$

The points $\boldsymbol{x}_{1i}, \boldsymbol{x}_{2i} \in \mathbb{R}^3$ in world space correspond to points on the respective constrained bodies such that $\Delta x_i = \boldsymbol{x}_{2i} - \boldsymbol{x}_{1i}$ gives the positional error. The vector $\boldsymbol{v}_{ri} \in \mathbb{R}^3$ is the desired relative velocity of $\boldsymbol{x}_{2i}$ with respect to $\boldsymbol{x}_{1i}$ and is analogous to the Jacobian constraint equation term $b(t)$ described in (§ 2.7.3.2). The term $\beta_i \in \mathbb{R}$ is a corrective factor such that $0 \leq \beta_i \leq 1$, similar to the Baumgarte stabilization factor [**65**] for Jacobian formulations.

Each discrepancy quadruple $\boldsymbol{d}_i$ is effectively an anchor point that limits the degrees of freedom between two bodies. Its components are computed depending on the current kinetic properties and the geometric properties of the bodies. The behaviour of a particular discrepancy-based constraint is thus dependent on the size of $\boldsymbol{c}$ and the computations performed to derive the discrepancy quadruples $\boldsymbol{d}_i$.

We choose to assign a corrective factor β_i to each discrepancy $\boldsymbol{d}_i$ rather than an overall corrective factor for the whole of $\boldsymbol{c}$. The motivation for this is that we may wish to prioritise some discrepancy quadruples over others within the same constraint for reasons of stability. A typical example is the modelling of an angular motor constraint in which the axle discrepancies are assigned higher priorities over the driving impulse discrepancies.

The proposed constraint model is inherently bilateral in that it does not evaluate to a valid or invalid state, unlike Jacobian-based constraints. Constraint unilateralism is hence achieved by the introduction or removal of constraints or of the discrepancies within them at appropriate times in the simulation.

2.7.5 Constraint Case Studies

In the following sections, we propose constructs for a number of common constraints, such as joints and contacts, based on the discrepancy-based constraint formulation proposed in (§ 2.7.4). Regrettably, we are unable to provide a thorough proof for these constructs due to limited time, and instead rely on geometrical intuition and empirical observation of the results. We defer this exercise as potential future work.

2.7.5.1 Collision and Contact

To validate our claim for a unified model we first consider the construction of a combined contact and collision constraint $\boldsymbol{c}_\mathrm{c} = [\boldsymbol{d}_{\mathrm{c}1} \quad \boldsymbol{d}_{\mathrm{c}2} \quad \dots \quad \boldsymbol{d}_{\mathrm{c}m}]$ from a reduced contact manifold as defined by (2.58a) and (2.58b) and given by the vector of m contact points

$$[(\boldsymbol{p}_1, \hat{\boldsymbol{n}}_1, d_1) \quad (\boldsymbol{p}_2, \hat{\boldsymbol{n}}_2, d_2) \quad \dots \quad (\boldsymbol{p}_m, \hat{\boldsymbol{n}}_m, d_m)]$$

Each contact point $(\boldsymbol{p}_i, \hat{\boldsymbol{n}}_i, d_i)$ may be converted to a contact discrepancy $\boldsymbol{d}_{\mathrm{c}i}$ using

$$\boldsymbol{d}_{\mathrm{c}i} = \left(\boldsymbol{p}_i - \frac{d_i}{2}\hat{\boldsymbol{n}}_i, \boldsymbol{p}_i + \frac{d_i}{2}\hat{\boldsymbol{n}}_i, \boldsymbol{v}_{\mathrm{r}i}, \beta_i\right) \tag{2.82}$$

where the discrepancy points are assumed to lie at equal distances $\frac{d_i}{2}$ from the contact point $\boldsymbol{p}_i$ along the contact normal $\hat{\boldsymbol{n}}_i$ such that the point discrepancy $\Delta p = d_i$ and $\boldsymbol{v}_{\mathrm{r}i} \in \mathbb{R}^3$ is the desired relative post-contact velocity of the contact point (Figure 45). To encourage rigidity in the collisions, we recommend relatively high values for the bias factor, such as $\beta_i = 0.8$. The lifetime of this constraint is short-term and matches the lifetime of the associated contact manifold.

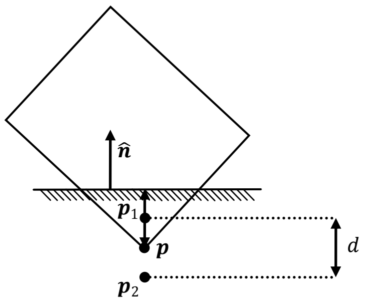


Figure 45 Discrepancy-based constraint model for collision

2.7.5.2 Ball-and-Socket Joint

A ball-and-socket, or *spherical* joint, binding two bodies constrains one body to freely rotate around the joint with respect to the other body, effectively removing three out of the twelve degrees of freedom of both bodies. This constraint requires two anchor points defined as fixed offsets $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{R}^3$ in the local frame of reference, such that their counterpart points $\boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{R}^3$ in world space coincide (Figure 46). Thus, the constraint equation in terms of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ is simply

$$\boldsymbol{p}_2 - \boldsymbol{p}_1 = \boldsymbol{0}$$

When the joint is invalidated, $\Delta p = |\boldsymbol{p}_2 - \boldsymbol{p}_1|$ gives the joint error and hence $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ may be used as discrepancy points.

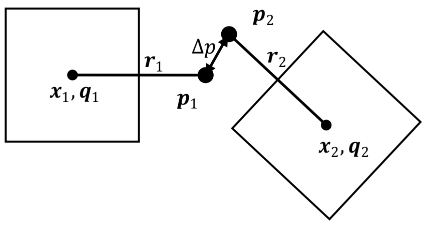


Figure 46 Discrepancy-based constraint model for a ball-and-socket joint

The points $\boldsymbol{p}_i$ may be expressed in terms of $\boldsymbol{r}_i \in \mathbb{R}^3$, body centroid positions $\boldsymbol{x}_i \in \mathbb{R}^3$ and orientations $\hat{\boldsymbol{q}}_i \in \mathbb{H}$ using

$$\boldsymbol{p}_i = \boldsymbol{x}_i + \hat{\boldsymbol{q}}_i \boldsymbol{r}_i \hat{\boldsymbol{q}}_i^{\,*}$$

as described in (§ 2.2.2).The desired relative velocity $\boldsymbol{v}_{\mathrm{r}}$ of $\boldsymbol{p}_2$ with respect to $\boldsymbol{p}_1$ follows from the derivative of the constraint equation

$$\boldsymbol{p}_2 - \boldsymbol{p}_1 = \boldsymbol{0} \Rightarrow \dot{\boldsymbol{p}}_2 - \dot{\boldsymbol{p}}_1 = \boldsymbol{0}$$

But $\boldsymbol{v}_{\mathrm{r}} = \dot{\boldsymbol{p}}_2 - \dot{\boldsymbol{p}}_1$ and hence $\boldsymbol{v}_{\mathrm{r}} = \boldsymbol{0}$. Therefore the ball-and-socket joint $\boldsymbol{c}_s$ may be defined in terms of one discrepancy point

$$\boldsymbol{c}_\mathrm{s} = [\boldsymbol{d}_\mathrm{s}] = [(\boldsymbol{x}_1 + \widehat{\boldsymbol{q}}_1 \boldsymbol{r}_1 \widehat{\boldsymbol{q}}_1{}^*, \boldsymbol{x}_2 + \widehat{\boldsymbol{q}}_2 \boldsymbol{r}_2 \widehat{\boldsymbol{q}}_2{}^*, \boldsymbol{0}, \beta)] \tag{2.83}$$

where we recommend a relatively strong correction factor $\beta = 0.8$.

2.7.5.3 Hinge Joint

A hinge joint, sometimes termed a *revolute joint*, binds two bodies around an axis that is fixed in the respective local frames of reference of each body, effectively removing five out of the twelve degrees of freedom of both bodies. The proposed discrepancy-based constraint formulation does not explicitly support axial anchors; however, we note that if two bodies are simultaneously bound by two non-coincident ball-and-socket joints, the bodies are constrained around an axis passing through the two joints (Figure 47).

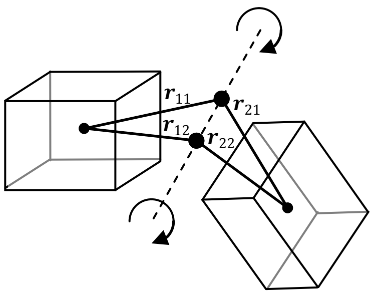


Figure 47 Discrepancy-based constraint model for a hinge joint

This observation suggests that a hinge constraint of the form $\boldsymbol{c}_\mathrm{h} = [\boldsymbol{d}_{\mathrm{s}1} \quad \boldsymbol{d}_{\mathrm{s}2}]$ may be constructed, where $\boldsymbol{d}_{\mathrm{s}1}$ and $\boldsymbol{d}_{\mathrm{s}2}$ specify anchor points on an axis passing through the points $\boldsymbol{r}_{11}, \boldsymbol{r}_{12} \in \mathbb{R}^3$ in the frame of reference of the first body, and $\boldsymbol{r}_{21}, \boldsymbol{r}_{22} \in \mathbb{R}^3$ in that of the second. Thus, $\boldsymbol{d}_{\mathrm{s}1}$ and $\boldsymbol{d}_{\mathrm{s}2}$ may be defined by

$$\boldsymbol{d}_{si} = (\boldsymbol{x}_1 + \widehat{\boldsymbol{q}}_1 \boldsymbol{r}_{i1} \widehat{\boldsymbol{q}}_1{}^*, \boldsymbol{x}_2 + \widehat{\boldsymbol{q}}_2 \boldsymbol{r}_{i2} \widehat{\boldsymbol{q}}_2{}^*, \boldsymbol{0}, \beta) \tag{2.84}$$

where we recommend a relatively strong correction factor $\beta = 0.8$. For improved stability, we further recommend that the axis anchor points be placed sufficiently far apart, typically the average width of the bodies involved along the direction of the axis.

2.7.5.4 Rigid Joint

Using similar arguments for the construction of a hinge constraint, we observe that when two bodies are bound simultaneously by three ball-and-socket joints (Figure 48), six out of twelve degrees of freedom are removed, effectively resulting in the bodies rigidly bound together under unified motion. Hence, by extending $\boldsymbol{c}_\mathrm{h}$, we define a rigid joint constraint $\boldsymbol{c}_\mathrm{r} = [\boldsymbol{d}_{\mathrm{s}1} \quad \boldsymbol{d}_{\mathrm{s}2} \quad \boldsymbol{d}_{\mathrm{s}3}]$ where $\boldsymbol{d}_{si}$ is defined as per (2.84) where, in this case, $\boldsymbol{r}_{11}, \boldsymbol{r}_{12}, \boldsymbol{r}_{13} \in \mathbb{R}^3$ and $\boldsymbol{r}_{21}, \boldsymbol{r}_{22}, \boldsymbol{r}_{23} \in \mathbb{R}^3$ are the respective anchor points in the local frame of reference of the first and second body.

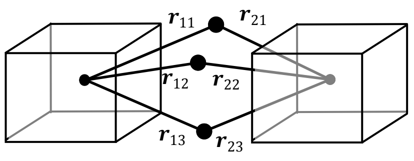


Figure 48 Discrepancy-based constraint model for a rigid joint

2.7.5.5 Angular Motor Joint

A velocity-based angular motor joint binding two bodies causes one body to rotate around a given axis at a given angular speed with respect to the other body. We observe that an angular motor must at the very least enforce a hinge constraint and we hence suggest that the discrepancy-based formulation should include the two axial discrepancy quadruples defined for the hinge constraint $\boldsymbol{c}_{\mathrm{h}}$ parameterised by $\boldsymbol{r}_{ij}$. Thus, the angular motor joint $\boldsymbol{c}_{\mathrm{a}}$ takes the form

$$\boldsymbol{c}_{\mathrm{a}} = [\boldsymbol{d}_{\mathrm{s}1} \quad \boldsymbol{d}_{\mathrm{s}2} \quad ...]$$

To enforce angular rotation, parameterised by $\omega \in \mathbb{R}$ around the axis, we derive an additional discrepancy $\boldsymbol{d}_{\mathrm{a}1}$ for enforcing the relative angular motion. We therefore compute a point $\boldsymbol{p}_1 \in \mathbb{R}^3$ by producing a vector $\boldsymbol{l}_1 \in \mathbb{R}^3$ orthogonal to the axis, from the midpoint of its anchor points in world space. For an more accurate computation of $\boldsymbol{p}_1$, we take into account positional drift, and hence we compute an average $\boldsymbol{a}_1, \boldsymbol{a}_2 \in \mathbb{R}^3$ of the axial anchor points $\boldsymbol{r}_{ij}$, transformed from the respective body frames of reference. We note that, for the motor constraint to hold true, the relative speed $v_r \in \mathbb{R}$ of the bodies at $\boldsymbol{p}_1$ must obey the equation

$$v_r = l\omega$$

where $|\boldsymbol{l}| = l$. The direction of v_{r} is orthogonal to both the axis direction $\widehat{\boldsymbol{a}} = \frac{\boldsymbol{a}_2 - \boldsymbol{a}_1}{|\boldsymbol{a}_2 - \boldsymbol{a}_1|}$ and the offset vector $\boldsymbol{l}_1$ of $\boldsymbol{p}_1$. Hence, the vector equivalent $\boldsymbol{v}_{r1} \in \mathbb{R}^3$ may be computed as

$$\boldsymbol{v}_{\mathrm{r}1} = \omega \widehat{\boldsymbol{a}} \times \boldsymbol{l}_1$$

The discrepancy driving the angular motor is thus given by

$$\boldsymbol{d}_{\mathrm{a}1} = (\boldsymbol{p}_1, \boldsymbol{p}_1, \omega \widehat{\boldsymbol{a}} \times \boldsymbol{l}_1, \beta) \tag{2.85}$$

where we recommend a relatively low correction factor $\beta = 0.4$ to give higher priority to the axial discrepancies $\boldsymbol{d}_{\mathrm{s}1}$ and $\boldsymbol{d}_{\mathrm{s}2}$. We note that in $\boldsymbol{d}_{\mathrm{a}1}$, there is no positional joint error, but a non-zero relative velocity component is specified as a means to drive the motor. The angular motor constraint may hence be defined as

$$\boldsymbol{c}_{\mathrm{a}} = [\boldsymbol{d}_{\mathrm{s}1} \quad \boldsymbol{d}_{\mathrm{s}2} \quad \boldsymbol{d}_{\mathrm{a}1}]$$

Although this definition is sufficient, we suggest that it is not sufficiently stable, and recommend the addition of at least another driving discrepancy $\boldsymbol{d}_{\mathrm{a}2}$ at a point $\boldsymbol{p}_2 \in \mathbb{R}^3$ reflecting $\boldsymbol{p}_1$ through the axis of rotation such that $\boldsymbol{l}_2 = -\boldsymbol{l}_1$ and $\boldsymbol{v}_{\mathrm{r}2} = -\boldsymbol{v}_{\mathrm{r}1}$ to balance the constraint. For further stability, we recommend two more driving discrepancies $\boldsymbol{d}_{\mathrm{a}3}$ and $\boldsymbol{d}_{\mathrm{a}4}$ at points $\boldsymbol{p}_3 \in \mathbb{R}^3$ and $\boldsymbol{p}_4 \in \mathbb{R}^3$ on a line orthogonal to both $\hat{\boldsymbol{a}}$ and $\boldsymbol{l}_1$, that is, $\boldsymbol{l}_3 = \boldsymbol{l}_1 \times \hat{\boldsymbol{a}} = -\boldsymbol{l}_4$, such that the motor is driven from four evenly spaced points lying on a plane orthogonal to the axis of rotation (Figure 49). Hence the enhanced angular motor constraint is specified by

$$\boldsymbol{c}_{\mathrm{a}} = [\boldsymbol{d}_{\mathrm{s}1} \quad \boldsymbol{d}_{\mathrm{s}2} \quad \boldsymbol{d}_{\mathrm{a}1} \quad \boldsymbol{d}_{\mathrm{a}2} \quad \boldsymbol{d}_{\mathrm{a}3} \quad \boldsymbol{d}_{\mathrm{a}4}] \tag{2.86}$$

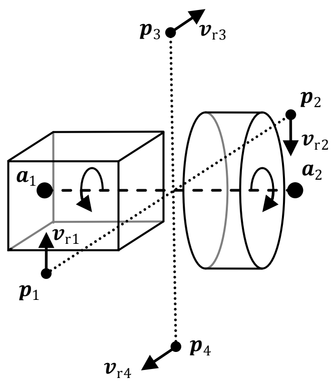

$\boldsymbol{p}_3$
$\boldsymbol{v}_{\mathrm{r}3}$
$\boldsymbol{p}_2$
$\boldsymbol{v}_{\mathrm{r}2}$
$\boldsymbol{a}_1$
$\boldsymbol{a}_2$
$\boldsymbol{v}_{\mathrm{r}1}$
$\boldsymbol{p}_1$
$\boldsymbol{p}_4$
$\boldsymbol{v}_{\mathrm{r}4}$

Figure 49 Discrepancy-based constraint model for an angular motor joint

2.7.5.6 Joint Limits

In practical applications, the motion allowed by a joint may be further restricted by angle ranges, distances or some bounded region. For instance, a ball-and-socket joint may be restricted to bend and rotate within a *reach cone*. In biomechanical simulations, the reach cone geometry of a ball-and-socket joint such as the human shoulder may be irregular, allowing a wider range of motion in one direction but a narrower range in another. Similarly, the angular range of a door hinge is restricted, at least due to its solid nature, which prevents it from turning through itself, unlike the hinge simulacrum proposed in (§ 2.7.5.3) which may indeed be used as an axle joint representation.

Like non-penetration contact conditions, joint limits are unilateral constraints that are enforced only when a joint is in a particular configuration. As a general approach for describing joint limits, we adapt an unlimited joint construct based on the discrepancy-based formulation, and define an allowed range of motion based on the joint geometry. We then extend the joint constraint with additional limiting discrepancy quadruples $\boldsymbol{d}_{\mathrm{l}i}$ whenever we detect joint motion beyond the allowed range.

As an example, we consider the ball-and-socket joint constraint $\boldsymbol{c}_{\mathrm{s}}$ and impose a joint limit in the form of a regular reach cone (Figure 50) parameterised by an angle $0 \leq \theta \leq \pi$.

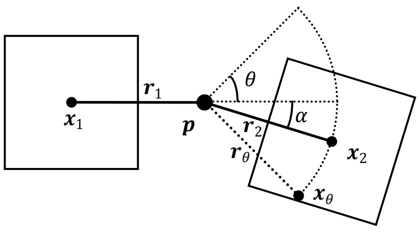


Figure 50 Discrepancy-based constraint model of a limited joint

Given the joint connection $\boldsymbol{p} \in \mathbb{R}^3$ in world space, we compute the cosine of the current angle $\alpha \in \mathbb{R}$ in terms of the body centroids $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^3$ using

$$\cos\alpha = -\frac{\boldsymbol{r}_1}{|\boldsymbol{r}_1|} \cdot \frac{\boldsymbol{r}_2}{|\boldsymbol{r}_2|}$$

where $\boldsymbol{r}_i = \boldsymbol{p} - \boldsymbol{x}_i$ and note that $\cos\alpha < \cos\theta \Leftrightarrow \alpha > \theta$ defines the condition where the joint exceeds the reach cone. During the simulation, if this condition is violated, we define a limiting constraint discrepancy $\boldsymbol{d}_\mathrm{l}$ as

$$\boldsymbol{d}_\mathrm{l} = (\boldsymbol{x}_\theta, \boldsymbol{x}_2, \boldsymbol{0}, \beta) \qquad (2.87)$$

where $\boldsymbol{x}_\theta \in \mathbb{R}^3$ is the limiting position that the centroid $\boldsymbol{x}_2$ can assume without exceeding the reach cone. We compute point $\boldsymbol{x}_\theta$ by producing a vector $\boldsymbol{r}_\theta \in \mathbb{R}^3$ of length $|\boldsymbol{r}_2|$ from $\boldsymbol{p}$, coplanar with points $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{p}$, such that $\boldsymbol{x}_\theta = \boldsymbol{p} + \boldsymbol{r}_\theta$ and

$$\cos\theta = \frac{\boldsymbol{r}_1}{|\boldsymbol{r}_1|} \cdot \frac{\boldsymbol{r}_\theta}{|\boldsymbol{r}_\theta|}$$

Therefore, we define the ball and socket-joint $\boldsymbol{c}_{\mathrm{s},\theta}$ limited by reach cone angle θ as

$$\boldsymbol{c}_{\mathrm{s},\theta} = \begin{cases} [\boldsymbol{d}_\mathrm{s}] & \cos\alpha \geq \cos\theta \\ [\boldsymbol{d}_\mathrm{s} \quad \boldsymbol{d}_\mathrm{l}] & \cos\alpha < \cos\theta \end{cases} \qquad (2.88)$$

2.7.5.7 Breakable Joints

A real world joint breaks when sufficiently large tensile or compressive forces are applied such that structural failure ensues in one or more joint components. Thus, an intuitive approach to simulate breakable joints between two bodies is to compute the sum contact forces $\boldsymbol{f}_\mathrm{c} \in \mathbb{R}^3$ applied by one body on to the other and remove the joint constraint if the magnitude of these forces exceeds a given limit $\varepsilon_\mathrm{f} \in \mathbb{R}$ specifying the robustness of the joint. The stress limit of the joint may be expressed as

$$|\boldsymbol{f}_\mathrm{c}| < \varepsilon_\mathrm{f}$$

Direct contact force computation requires knowledge of all internal and external forces, which may not feasible, for instance, when using impulse-based systems. However, over a small time-step $\Delta t \in \mathbb{R}$, the contact forces may be approximated by

$$\boldsymbol{f}_\mathrm{c} \cong \frac{\boldsymbol{j}_\mathrm{c}}{\Delta t}$$

where $\boldsymbol{j}_\mathrm{c} \in \mathbb{R}^3$ is the sum impulse at the joint. Hence, the stress limit may be expressed in impulse-based terms as $\boldsymbol{j}_\mathrm{c} < \varepsilon_\mathrm{f}\Delta t$. This approach assumes an impulse-based constraint solver, which conflicts with our goal for a solver-agnostic constraint representation. The discrepancy-based constraint formulation does however allow for the computation of positional joint errors, and the stress limit may accordingly be expressed in positional terms by noting that

$$\boldsymbol{j}_\mathrm{c} \cong \frac{g}{\Delta t}\Delta \boldsymbol{p}$$

where $g \in \mathbb{R}$ is a scalar function of the linear and angular mass properties and the geometry of the joint, with a derivation similar to the denominator in (2.64). Assuming g constant over the interval spanned by Δt, the stress limit may thus be expressed in terms of the joint error by

$$\Delta \boldsymbol{p} < \varepsilon_\mathrm{f} \frac{\Delta t^2}{g}$$

If the time step Δt is known to be fixed throughout the simulation, the force-based stress limit condition, expressed in terms of ε_f, may be replaced by a position-based stress limit, involving the joint error limit ε_p such that it may be expressed simply as

$$\Delta \boldsymbol{p} < \varepsilon_\mathrm{p} \tag{2.89}$$

where ε_p may be derived by empirical observation, or analytically by an approximation such as

$$\varepsilon_\mathrm{p} \cong \varepsilon_\mathrm{f} \frac{\Delta t^2}{g}$$

The discrepancy-based constraint formulation defined in (2.81a) and (2.81b) may hence be extended to

$$\boldsymbol{c} = \left([\boldsymbol{d}_1 \quad \boldsymbol{d}_2 \quad \dots \quad \boldsymbol{d}_m], \varepsilon_\mathrm{p}\right) \tag{2.90}$$

where the constraint $\boldsymbol{c}$ is deemed active as long as none of the discrepancies $\boldsymbol{d}_i$ break condition (2.89).

2.7.6 Constraint Solvers

In this section we briefly describe a possible constraint solver implementation for the model defined in (§ 2.7.4). The proposed solver is of a sequential nature and operates in a matter very similar to the Jacobian constraint solver discussed in (§ 2.7.3.2). The solver essentially iterates for a given number of times $k \in \mathbb{N}$, processing the constraint discrepancies $\boldsymbol{d}_{ij}$ of all the constraints $\boldsymbol{c}_i$ in the system, solving a fraction $\frac{1}{k}$ of the positional and velocity discrepancies, parameterised by an overall correction factor $0 \leq \beta \leq 1$ that is modulated with the discrepancy-level factor β_{ij} of $\boldsymbol{d}_{ij}$.

We suggest that LCP-based solvers are also feasible with the proposed constraint model. However, due to time limitations, we defer this research area as potential future work.

Chapter 3
Engine Design and Implementation

In this chapter we describe the specification and design of the Gravitas physics engine with a focus on the architectural, technical, theoretical and pragmatic decisions that led to its present form. In particular, we refer to the theory surveyed in Chapter 2 to justify the engine's design from a functional point of view. The resulting implementation is a moderately large codebase with a very extensive application programming interface (API) reflecting the minutiae of all the functionality provided by the engine. For reasons of brevity, we limit our description of the constituent components at the conceptual level.

3.1 Design Principles

The engine is architected according to a number of desirable principles, applicable to software design, and also specific to the functional requirements of the engine. These principles may be summed up under a number of headings, namely: reusability, flexibility, performance, robustness, plausibility, visibility, modularity, abstraction, extensibility, consistency. From the software design perspective, the architecture is based on object orientation paradigms [**1**] and the application of software design patterns [**2**].

3.1.1 Reusability and Flexibility

One of the primary aims of Gravitas is to enable software applications with physics-based animation functionality. Gravitas is thus intended as a third party software module that may be integrated with a software application at the development stage through its application programming interface (API). The design of the engine's API was driven by the requirement to support a wide range of applications. This was accomplished through the provision of many configuration options and the ability to tailor the physics engine, through available or custom-made pluggable modules, or through application-provided extensions. One of the motivations behind the choice of C++ [**66**] as the implementation language for the physics engine is to maximise portability of the engine across different platforms.

At its core, the Gravitas physics engine consists of a number of abstract interfaces for its various components. This enables the API user to customise all the pertinent aspects of a physical simulation environment by choosing the appropriate implementation for each interface related to a given aspect. Examples include the choice of body geometry used in the simulation, the numerical kinetic integrator used to drive forward the motion of the bodies, the collision detection and constraint solvers used in the simulation, and the spatial representation used to perform broad-phase collision pruning.

3.1.2 Performance

The Gravitas physics engine is intended for real-time applications such as interactive computer simulations and games, thus, its implementation necessitated a programming language supporting sophisticated constructs for tackling a complex project while still yielding highly efficient executable code. The C++ language is thus the natural choice for the endeavour, which incidentally, is considered the *de facto* language for such applications.

Regardless of language, the design of the engine is a balancing game between the realisation of a flexible and extensible API, versus the provision a framework that encourages efficient implementations of its interfaces. The engine makes extensive use of abstraction layers, such as pure abstract interfaces and a plug-in framework. Although these levels of indirection do incur a slight overhead, we feel that it pales to insignificance vis-à-vis the gains resulting from appropriate use of high-level data processing algorithms.

3.1.3 Robustness

A simulation is a computationally expensive process involving thousands, or potentially millions, of integer and float point digital operations. Unlike analytical computing, which offers several techniques for working around numerical singularities, extra care is required when dealing with limited precision floating point arithmetic to avoid numerically unstable methods. A typical example is the multiplication by very large numbers, or conversely, a division involving very small divisors, which may lead to numerical overflow errors. Another problem entails the comparison of floating point values. A series of digital floating point computations is likely to introduce minor numerical errors in the result due to the limited precision of digital computation. This makes precise comparisons of such values unreliable. Many of these issues may be mitigated using tolerance values and *interval arithmetic* [**67**]. These approaches are used throughout the produced codebase to minimise numerical instability issues.

3.1.4 Plausibility

Modelling of the motion of bodies is governed by the theory and laws of classical mechanics as described in Chapter 2. These idealised models have the capability of simulating perfectly rigid bodies with perfectly smooth surfaces. This idealisation is often counterproductive in terms of the desired results [**68**]. A typical example is dropping perfectly shaped rigid spheres on top of each other on a perfectly horizontal ground plane. The first sphere dropped will bounce on the ground along a vertical axis, eventually settling at the original impact point. Subsequent spheres will bounce on the first sphere, resulting in eventual stacking of the spheres on top of each other, in perfect balance. This behaviour, while correct with respect to the model used, is perceived as unrealistic or implausible. Hence, the physics engine is designed to accommodate imperfections within the simulation model. Such measures include velocity damping coefficients and contact surface perturbation to model uneven surface microstructure.

Another facet of plausibility, particularly in the context of real-time applications such as computer games, is the trade-off between the accuracy of the models versus the performance of the simulation and the quality of the results. Physical simulation in such applications is typically allocated a relatively small portion of the available processing resources, and must contend with other processing tasks, such as graphics, networking and artificial intelligence. Hence, the physical models employed must be simple enough to fit the processing budget, yet sufficiently sophisticated to yield results that are perceived as realistic by the intended audience. To this end, the physics engine is designed to allow for different implementations for every aspect of the simulation via the interface abstraction and plug-in mechanisms. This enables the API user to mix and match components, each characterised by specific levels of performance and accuracy, to suit the application needs.

3.1.5 Visibility and Modularity

The physics engine was designed as a self-contained module independent of its application, and in particular, the visualisation technology used to present simulation states to its users. On the other hand, there is a clear requirement for tighter coupling with external modules, such as passing notifications of collisions and joint breakages. Another requirement, particularly relevant when building and debugging new plug-ins for the engine, is the ability to render internal debugging and tracing information, termed *instrumentation*, during the simulations. All these requirements are met through the use of call-back mechanisms implemented using the *Observer* design pattern [**69**]

The requirement for flexibility in the design of the physics engine conflicts with the desire for decoupling the various components constituting the engine, into self contained modules. This need for flexibility is due to the wide range of simulation techniques available, which discourage many assumptions on the physics modelling pipeline and the degree of interaction between the components. The design of the engine thus attempts to provide interfaces that are sufficiently generic to accommodate as many techniques as possible.

3.1.6 Abstraction and Extensibility

Within Gravitas, abstraction takes the form of pure abstract interfaces for the components of a simulation, enabling their interoperation regardless of the underlying implementations used. Examples of such interfaces include those for the definition of body geometry, collision algorithms, spatial representations, numerical integrators and constraint solvers. The notion of abstract interfaces is further exploited within the plug-in mechanism. This enables components, packaged in dynamic libraries, such as the Dynamic Link Library (DLL) specification for Windows, to be loaded into the engine and used at run-time.

3.1.7 Consistency

The modular organisation of the engine follows a common theme where all sub-systems are handled by centralised manager modules. The services offered by each module share a degree of commonality with all other modules such that familiarisation with one sub-system facilitates the understanding of the other sub-systems. Each module generally provides component registration and query services, with access to components provided through abstract interfaces. The components provide functionality relating to their specific sub-system via abstract interfaces.

3.2 Gravitas Organisational Structure

The Meson: Gravitas physics engine is part of a collaborative effort (Figure 51) involving this work and the realisation of the Vistas visualisation engine [**3**] and the Cuneus scripting engine [**8**] constituting a unified simulation platform named Meson, supported by a common framework (§ 3.3.1). We envisage this platform as a tool for constructing interactive simulations and computer games powered by the graphics rendering capabilities of Vistas, coupled with realistic physical environments and script-driven artificially intelligent agents and events powered by Cuneus.

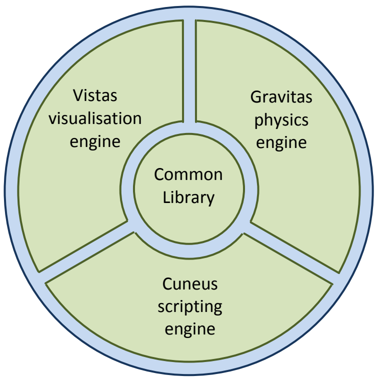


Figure 51 The Meson simulation platform

The Gravitas physics engine consists of a number of modules organised in a hierarchical structure. At the highest level, the engine, which is represented by the class `GravitasEngine`, exposes a number of centralised manager modules (Figure 52), each responsible for a specific sub-system through a set of services for external or cross-module use. The engine maintains a reference to exactly one instance of each manager module by enforcing a *Singleton* pattern [**70**].

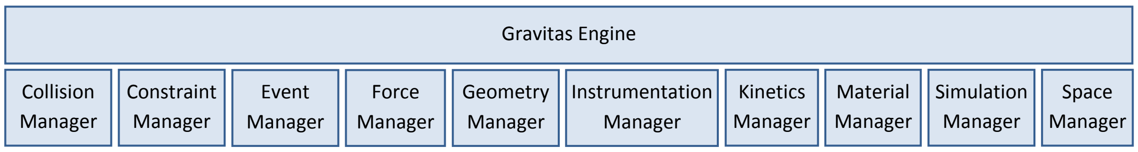


Figure 52 High-level view of the Gravitas engine

A component provided by a manager module may represent an algorithm or technique, or may represent an instantiated data structure or object. Algorithmic components are inherently behavioural and only one instance per algorithm type is allowed in the engine by enforcing a *Singleton* pattern. In this case, the component implementations are abstracted by a behavioural interface via a *Strategy* pattern [**71**]. Data-centric components on the other hand tend to represent simulation entities and accordingly, multiple instances may be required.

Such components are instantiated through factory modules registered with a corresponding manager, and are based on the *Abstract Factory* pattern [**72**].

3.3 Foundations

In this section, we provide a very brief survey of the internal and external codebase supporting the Gravitas physics engine.

3.3.1 Common Library

The Common library is a joint effort to provide a common set of cross-platform services for the Meson Simulation Platform consisting of the Gravitas physics engine, the Vistas visualisation engine [**3**] and the Cuneus scripting engine [**8**]. The library features templated collections, event support, input/output file and stream support, data logging, mathematical constructs and solvers, memory management, smart pointers, a plug-in mechanism, a custom property mechanism, random number generation, regular expressions, serialisation, system services, and advanced text and XML services.

Of particular relevance to Gravitas are the mathematical constructs such as vectors, points, matrices, quaternions and intervals, provided as templated classes **`TVector3<TReal>`**, **`TPoint3<TReal>`**, **`TMatrix3<TReal>`**, **`TMatrix4<TReal>`**, **`TQuaternion<TReal>`** and **`TInterval<TReal>`**. Also crucial to the performance of the simulation algorithms are the templated list, set and map collections, such as **`TArrayList<TElement>`**, **`THashSet<TElement>`** and **`TTrieMap<TElement>`** to name a few. Besides eliminating a dependency on a third party collection library, these collection implementations are specifically optimised the high performance requirements of real-time simulation and animation.

A lot of effort has been dedicated to the realisation of the Common library; however, for reasons of scope and brevity, we limit ourselves to this short passage and make references to this section as appropriate in subsequent sections.

3.3.2 Gravitas Base Class

Virtually all classes within the physics engine derive from a base hybrid-interface class named **`IGravitasEntity`**. This interface provides a number of services, including component identification, a custom property system, and instrumentation rendering.

Within Gravitas, component associations are generally handled using unmanaged or managed pointers. However, the component identification mechanism finds application in disk-to-memory serialisation mechanisms whereby components qualified by string-based identifiers on disk are eventually associated in memory using pointers.

The custom property system works in tandem with the plug-in mechanism provided by the Common library (§ 3.3.1). It exposes metadata about the names and types of properties acceptable by a given component. This facilitates the construction of graphical tool applications that allow a designer to specify values to the properties of a specific Gravitas component.

The instrumentation rendering capability of **`IGravitasEntity`** allows any component extending the interface to render graphical information, or instrumentation, about its internal state, as described in (§ 3.9).

3.4 Collision Manager

The Collision manager module (Figure 53), represented by class **`CollisionManager`**, maintains a registry of narrow-phase collision detection algorithms abstracted by the Strategy pattern-based interface **`ICollisionDetector`**. The interface in turn defines a set of collision detection queries for two given types of geometry. Internally, the collision manager maintains a matrix-based mapping where each algorithm instance is associated with a row index and column index corresponding to the geometry type. As the geometry type is specified by a character string, the manager also maintains a string to index mapping. The motivation for these map constructs is to speed up run-time queries for a specific collision detector given two geometry instances of common, or different, type.

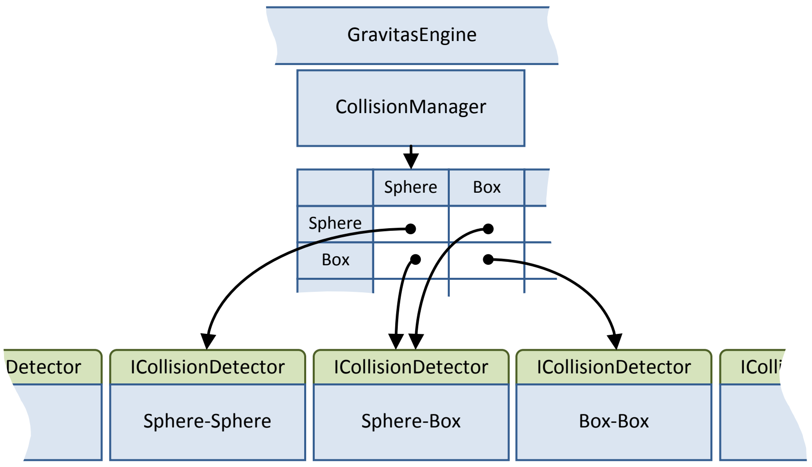


Figure 53 Collision manager

3.4.1 Collision Detectors

The purpose of the **`ICollisionDetector`** interface is to enable the abstraction of collision detection algorithms tailor-made for specific types of geometry. The interface defines three methods corresponding to the three types of geometrical queries as described in (§§ 2.5.1 – 2.5.3), namely, intersection testing, TOI estimation and contact manifold generation.

3.4.1.1 Intersection Tests

The geometry of two given bodies being tested for intersection (Figure 54) is specified by two references to instances of the abstract interface **`IGeometry`** (§ 3.8.1). As per the motivating discussion in (§ 2.5) the placement of the geometry is given by a relative transform expressing the placement of the second geometry with respect to the first. The transform is embodied in the **`Transform`** class (§ 3.8.2) implementing a quaternion-based transform as described in (§ 2.2.2).

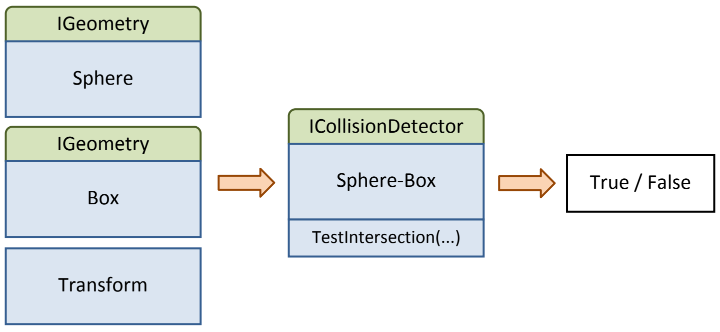


Figure 54 Intersection testing

3.4.1.2 Time of Impact Tests

Like the intersection testing query, the TOI estimation query (Figure 55) accepts two geometry instances and a relative transform parameter. In addition, the query accepts an upper bound for the time estimate and a writable parameter to contain the computed estimate. Both the intersection and the TOI tests are predicate methods that return a Boolean true value if the tests succeed, or false otherwise.

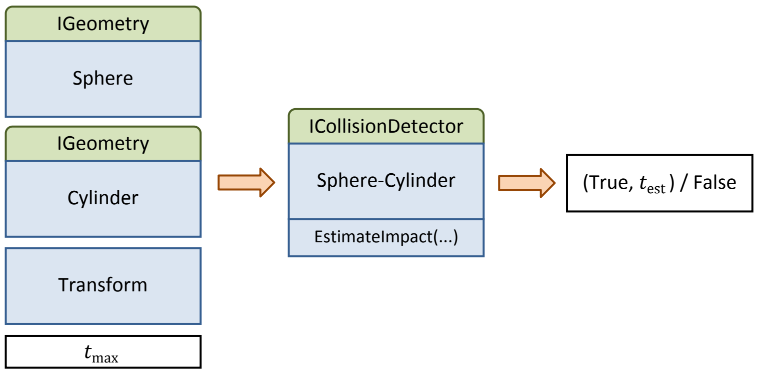


Figure 55 Time of impact estimation

3.4.2 Contact Manifold Generation

The contact manifold generation method (Figure 56) accepts the geometry of the bodies and a relative transform together with a data structure representing the reduced contact manifold as described in (§ 2.5.3.1), represented by the class **`ContactManifold`**, that is populated by the method. This contact manifold class contains references to the bodies involved in the contact or collision, and a list of contact points represented by the structure **`ContactPoint`** analogous to the mathematical definition (2.58b), consisting of a contact position, normal vector and penetration depth. In addition, the contact point structure includes the relative velocity of the contact point as described in (§ 2.6.2.1) to facilitate collision resolution models.

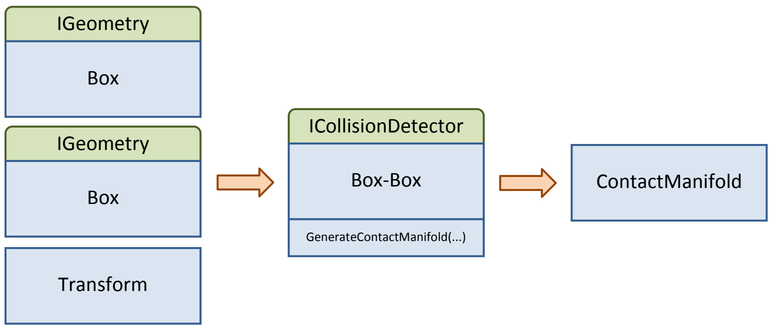


Figure 56 Contact manifold generation

3.4.3 Collision Filter

A simulation environment (§ 3.12.2) contains an optional reference to a collision filtering module, represented by class `CollisionFilter`. A collision filter instance is essentially a rule engine (Figure 57) that allows the API user to filter out specific collisions before they are processed by the constraint solver. As a motivating example, the simulation of an avalanche consisting of rocks tumbling against a cliff wall may be optimised by allowing only collisions between the rocks and the wall but not between each other.

The filtering rules provided by this module may be applied at individual body (§ 3.14.1) level or at collision group (§ 3.4.4) level. The filtering criteria supported are:

- any collision with a given body
- specific pairs of bodies
- collisions involving at least one body that is a member of a given collision group
- collisions involving bodies with respective membership in two given groups

The filter also supports exceptions analogous to the rules above. This enables the API user to specify generic rules with specific exceptions. A default catch-all rule is also available. This applies when no other rule is triggered by a given body pair. This rule is the final arbiter for a filtering decision.

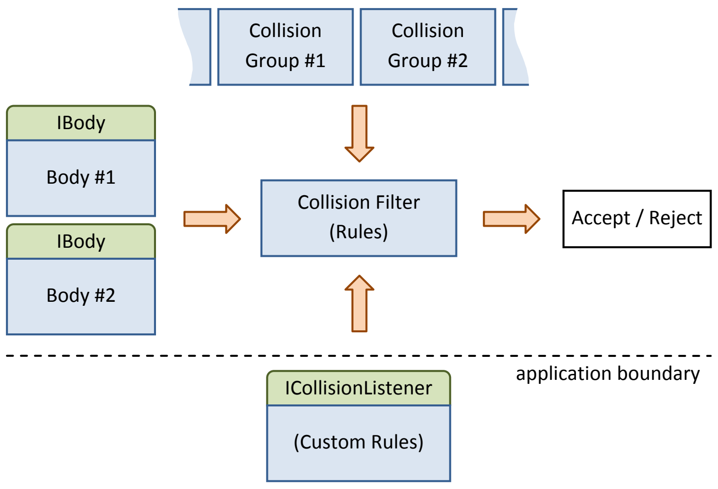


Figure 57 Collision filtering

If a simulation environment's collision filter reference is not set, all collisions are processed by default. Alternatively, if collision filtering events are enabled, any subscribed listeners may perform custom filtering (§ 3.6.3).

3.4.4 Collision Groups

To facilitate the specification of collision filtering rules and exceptions, collision groups, represented by class `CollisionGroup`, provide a means to define rules applicable to a group of bodies, rather than an individual body (§ 3.14.1). A rule or exception involving a group affects all bodies that are members within it. For added flexibility, a body is allowed to have simultaneous membership in more than one collision group.

3.5 Constraint Manager

The constraint manager module, represented by the class `ConstraintManager`, is responsible for the registration of constraint factories and constraint solvers. The constraint factories and solvers respectively implement the Abstract Factory pattern [**72**] and the Strategy pattern [**71**] via appropriate abstract interfaces.

3.5.1 Constraints

Constraint instances are created from implementations of the `IConstraintFactory` interface maintained by the constraint manager (Figure 58). Constraints in Gravitas are based on the discrepancy based formulation (2.81a) and (2.81b) we proposed in (§ 2.7.4). The behaviour of a constraint is abstracted by the interface `IConstraint`, corresponding to (2.81a), that defines methods for providing appropriate point discrepancies (2.81b) using lists of the structure `ConstraintDiscrepancy`. The constraint interface also allows the API user to specify a joint error threshold to support breakable joints (§ 2.7.5.7). The constraint discrepancy structure, in line with its mathematical representation, contains a pair of discrepancy points, a desired relative velocity vector and a corrective factor. Each constraint implementation is parameterised by a number of properties using the custom property mechanism (§ 3.3.2). For instance, a ball-and-socket joint expects a connection point between two bodies, whereas an angular motor expects two points defining an axle and a value for the desired relative angular speed.

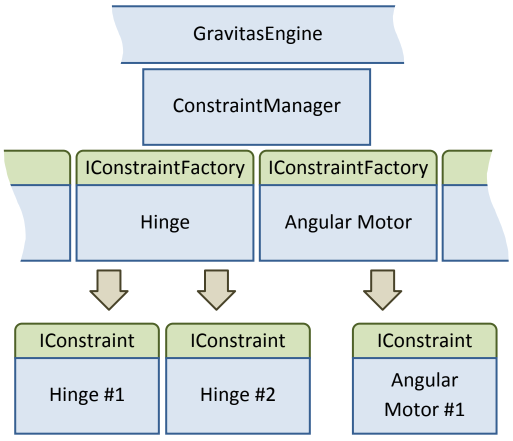


Figure 58 Constraint management

As collision and contact constraints are required in virtually all rigid body dynamics simulations, an inbuilt implementation is provided within the engine, and is represented by the class **`ContactConstraint`**. The class provides a means for converting a reduced contact manifold into a constraint as discussed in (§ 2.7.5.1).

3.5.2 Constraint Solvers

The constraint manager maintains a registry of constraint solver implementations abstracted by the **`IConstraintSolver`** interface (Figure 59). Simulations employ a specific constraint solver through a reference stored in their associated environment (§ 3.12.2). Solvers may also be parameterised via the custom property mechanism (§ 3.3.2). For instance, an iterative solver may be configured for a specific number of iterations.

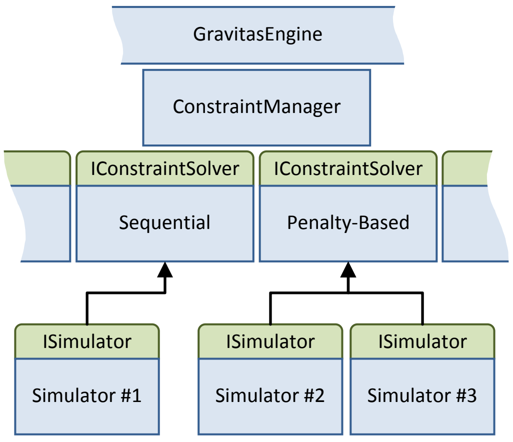


Figure 59 Constraint solver management

3.5.3 Constraint Batches

To improve computational performance, constraint solvers operate with islands, or partitions, of related bodies (§ 2.6.7.3). In Gravitas, such partitions are represented by instances of the structure **`ConstraintBatch`**, where each batch is defined in terms of the constraints relating a number of bodies together. The batching or partitioning process is performed by the constraint batch aggregator module (Figure 60), represented by class **`ConstraintBatchAggregator`**.

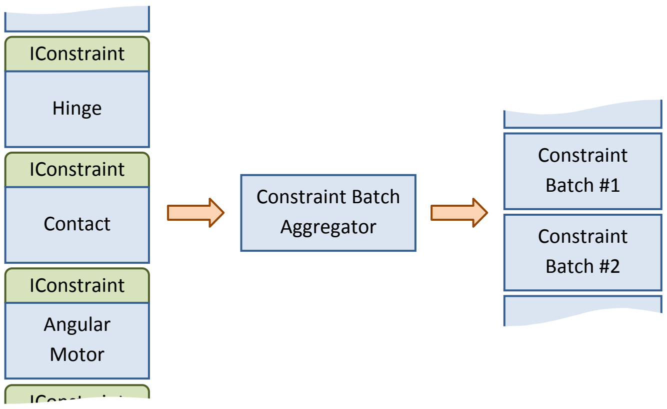


Figure 60 Constraint batching

The batching algorithm employed in Gravitas is an improved variation of the process summed up by the definition (2.80). The constraints are processed in sequence and matched against the current working sets, of which there are initially none. If no matching set is found, a new one is created to hold the constraint. If exactly one matching set is found, the constraint is inserted within it, otherwise, all matching sets bridged by the constraint are combined together into a single set. The algorithm terminates when all constraints are processed, at which point, the sets cannot be amalgamated any further, effectively resulting in one or more partitions of the current constraints.

3.6 Event Manager

The event manager module (Figure 61), represented by class **`EventManager`**, provides event notification services to other modules within the physics engine and enables both internal modules and external application code to subscribe to specific types of events. A set of flags allow the API user to control which event types are enabled, or otherwise. The current event types supported are related to activity management, collision occurrences, collision filtering, constraints and material mixing.

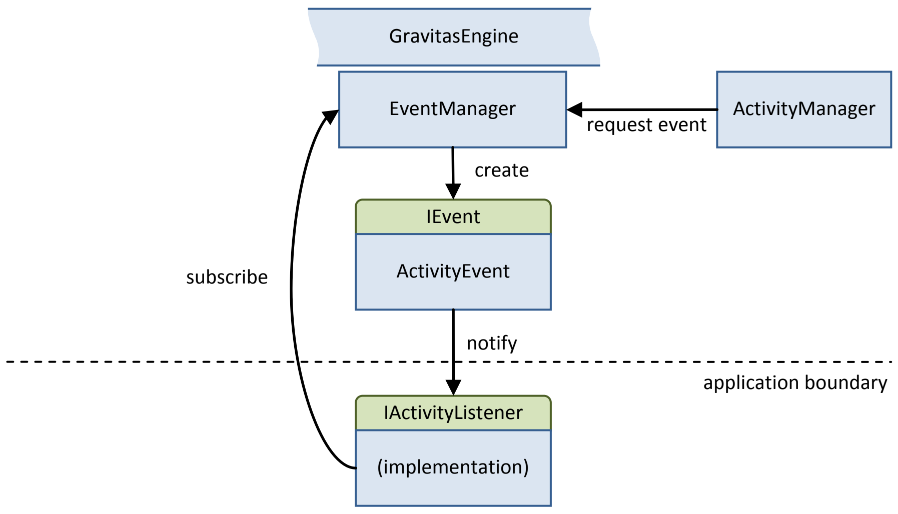


Figure 61 An example of the event manager servicing another module

All event types implement a partially abstract interface **`IEvent`** that provides reference counting for use with smart pointers as outlined in (§ 3.3.1). Event notifications are captured by other modules internal or external to Gravitas using an Observer pattern [**69**]. For every type of event, a corresponding *listener* interface is provided such that it may be implemented by any module that wishes to subscribe to the associated event type.

3.6.1 Activity Events

An activity event, embodied by class **`ActivityEvent`**, with corresponding listener interface **`ActivityListener`**, specifies a reference to a body, and an enumerated type that specifies if the event is signalling suspension or resumption of the body's kinetic activity. An activity event request is submitted by the activity manager module (§ 3.10.4) to the event manager that, in turn, creates the event and notifies all the activity listeners subscribed to it.

3.6.2 Collision Events

A collision event, represented by class **`CollisionEvent`**, with corresponding listener interface **`CollisionListener`**, specifies a reference to a contact manifold (§ 3.4.2), and an enumerated type that specifies if the event is signalling a starting, ongoing or terminating collision between two bodies.

Collision event requests are submitted by a collision tracker module represented by class **`CollisionTracker`** that in turn, interacts with the collision management system. The collision tracker essentially samples the current collisions, detecting changes such as the introduction of new collisions or absence of collisions with respect to an earlier sample. A timing parameter regulates the sampling rate, by default set to 10Hz, to prevent excessive event traffic.

On receipt of an event request, the event manager creates the event and notifies all the collision listeners subscribed to it.

3.6.3 Collision Filtering Events

A collision filtering event, embodied by class `CollisionFilterEvent`, with corresponding listener interface `CollisionFilterListener`, specifies references to two bodies (§ 3.14.1) in contact, and a flag specifying whether the collision should be allowed or filtered out.

Collision filtering events effectively provide a call-back mechanism allowing a listener to override the filtering rules configured within the collision filter module (§ 3.4.3) associated with a simulation environment. If collision filtering events are enabled, for every pair of bodies processed, the collision filter submits an event request to the event manager that, in turn, creates the event and notifies all the collision filter listeners subscribed to it. Each listener is given the opportunity to set the filtering flag according to some custom rule. Following the listener notifications, the collision filter checks the event's filter flag for the custom filtering result.

3.6.4 Constraint Events

A constraint event, embodied by class `ConstraintEvent`, with corresponding listener interface `ConstraintListener`, specifies a reference to a constraint (§ 3.5.1), and an enumerated type that specifies the constraint event type. A constraint event request is submitted by a simulator instance (§ 3.12.1) to the event manager that, in turn, creates the event and notifies all the constraint listeners subscribed to it.

To date, the only type of constraint event supported is a joint breakage. Extension of this event type is reserved as potential future work.

3.6.5 Material Events

A material event, embodied by class `MaterialEvent`, with corresponding listener interface `MaterialListener`, specifies references to two materials of bodies involved in contact or collision, and an additional reference to a combined material.

Material events act as a call-back mechanism allowing a listener to override the default material mixing computations (§ 3.11.23.4.3) performed by Gravitas. If material events are enabled, for every pair of materials processed, an event request is submitted to the event manager that, in turn, creates the event and notifies all the material listeners subscribed to it. Each listener is given the opportunity to perform custom material mixing computations. Following the listener notifications, the combined material reference is checked for the custom computations.

3.7 Force Manager

The force manager (Figure 62), embodied by the class **`ForceManager`**, is a central registry for force factories and force instances. The force factories follow the Abstract Factory pattern [**72**] via an interface **`IForceFactory`**. Each force factory defines a specific type of force, such as a linear gravity field, or a fluid drag force. Individual force instances created from the factories may also be registered with the force manager for access via a string-based identifier.

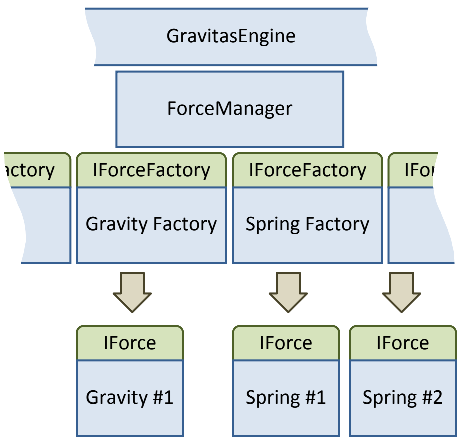


Figure 62 Force manager

3.7.1 Forces

Forces are represented by an interface **`IForce`** that abstracts the underlying implementation, effectively acting as a Strategy pattern [**71**]. A force instance defines a vector for the force value and an application point specified in a local frame of reference. Forces may be parameterised via the custom property system of the base interface **`IGravitasEntity`** described in (§ 3.3.2). In addition, when a force is bound to a body (§ 3.14.1) it operates on the body's mass, kinetic, and geometric properties. For instance, a gravity force may be implemented such that it accepts an acceleration vector due to gravity. When such a force is bound to a body, it computes a force as the product of the acceleration property and the body's mass.

3.7.2 Force Accumulators

A force accumulator in Gravitas, represented by class **`ForceAccumulator`**, is a bridging entity that binds the mass, kinetic and geometric properties of a body to a number of forces (Figure 63). The force accumulator, as its name implies, provides functionality for accumulating several forces applied on a body into an overall sum force or torque. In addition, the accumulator is also provides vector-time functor instances representing the sum linear and angular accelerations resulting from the forces applied on the associated body.

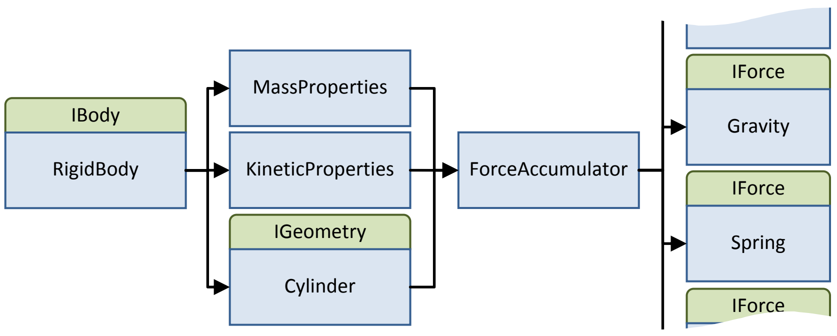


Figure 63 Force accumulators

3.8 Geometry Manager

The geometry manager, represented by class **`GeometryManager`**, is a registry for geometry factories and geometry instances (Figure 64). The Abstract Factory pattern [**72**] is employed to allow the API user to register implementations of the factory interface **`IGeometryFactory`**. The interface defines a creation method such that each factory implementation is responsible for the instantiation of specific types of geometry, represented by the interface **`IGeometry`**. Geometry instances created by the factories may also be registered with the manager for subsequent access via a string-based qualifier.

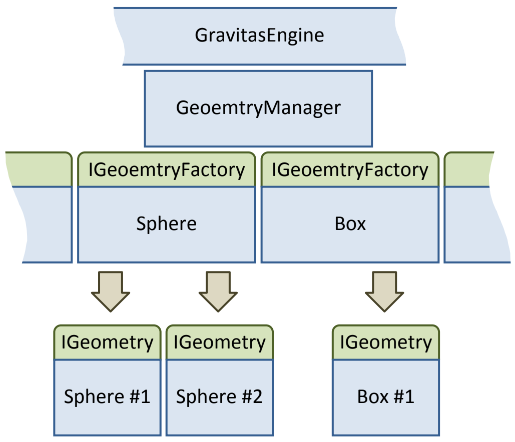


Figure 64 Geometry manager

3.8.1 Body Geometry

A Strategy pattern [**71**] is employed for representing the geometric characteristics of bodies in a simulation. The geometry of a body is represented by the partially abstract interface **`IGeometry`** that provides a number of methods related to type assignment, mass property calculation, by bounding volume enclosure and ray intersections. A geometric instance is analogous to a volume or region in space defining the overall shape of a body, as described in (§ 2.5.1).

The collision manager indexes collision detection algorithms by converting geometry type identifiers in string from to integers and using these for fast indexing into a matrix (§ 2.5). The geometry interface provides methods for maintaining local mappings of these type identifiers to assist the collision manager during collision detection algorithm retrieval.

The interface also defines methods for computing the volume and the distribution tensor (2.32) of a geometric instance, assuming this is bounded.

To facilitate mid-phase collision detection, all geometry instances are required to define methods for computing tight bounding spheres, axis-aligned boxes or oriented boxes, provided that the geometry is unbounded.

The geometry interface is also required to implement ray intersection testing methods, which are assumed in the local frame of reference of the geometry. This functionality is used by the ray intersection tests (§ 3.13.2) provided by the space implementations.

Each instance of an **`IGeometry`** implementation may be parameterised through the custom property system provided by its parent interface **`IGravitasEntity`** (§ 3.3.2) such that it is possible to define geometry of the same fundamental type but different dimensions, such as spheres of different radii or boxes with different extents. A particular instance may be shared by multiple bodies (§ 3.14.1) as long as their geometry is required to be of the same fundamental type and dimensions.

3.8.2 Spatial Transform

Spatial transforms are based on the quaternion formulation (2.5) as described in (§ 2.2.2) and are embodied by the structure **`Transform`**. The transform structure provides methods and overridden operators for performing transform composition, computing inverse transforms and changing basis with respect to another transform representing a frame of reference. Additionally, the transform provides methods for transforming individual points and point lists. A method is also provided for extracting an equivalent homogenous 4x4 matrix as per the formulation (2.1) as described in (§ 2.2.1). This is required for converting transforms to a format amenable to visualisation technologies that operate with homogenous matrix transformations.

3.8.3 Bounding Volumes

The bounding volumes supported by Gravitas are the sphere, the axis-aligned box and oriented box. These are the building blocks for constructing algorithms and data structures for broad-phase and mid-phase collision detection.

All three bounding volumes implement the interface `IBoundingVolume`, defining methods for point, ray and triangle intersection tests, for testing intersection with other bounding volumes, dynamic intersection tests assuming a relative translational volume sweep, closest point tests, spatial transformations and axial projections. These geometric queries are used to construct higher-level collision detection algorithms, such as those involving SAT's, BVH's, and TOI estimation.

Bounding volumes in Gravitas are fully specified in world space, and hence their properties include the placement, and where applicable, the orientation of the volume. A bounding sphere, implemented by class `BoundingSphere`, is specified by a sphere centre and a radius. An axis-aligned bounding box, implemented by class `BoundingAxisAlignedBox`, is specified by minimum and maximum point extents, representing the extreme vertices of the box. An oriented bounding box, implemented by class `BoundingOrientedBox`, is specified by a centroid, a set of three orthonormal axis vectors and three scalar extents along each axis.

3.8.4 Geometric Primitives

Complementing bounding volumes are a number of elementary geometric primitives used in higher level geometric queries. For instance, a variant of the GJK method for determining point containment in a convex polyhedron (§ 2.5.1.2) constructs progressively converging point, line, triangle or tetrahedron simplices using the constructs we describe in this section.

3.8.4.1 Line Segments

A line segment in Gravitas is represented by the structure `LineSegment` that is defined in terms of a starting and an ending point. The structure provides methods to compute length, squared length, a relative vector offset of the endpoints, a direction unit vector, midpoint, and perform closest point and distance queries from other points or line segments. The line segment may also be inverted or transformed using the `Transform` structure (§ 3.8.2).

3.8.4.2 Rays

A directed ray is represented by the structure `Ray` and is defined in terms of a source position for the ray and a unit direction vector. The ray structure provides methods to orient the ray towards a given point, to transform it using a `Transform` construct (§ 3.8.2) or a three dimensional matrix, compute distance and closest point queries to other points, and to compute a point along the ray parameterised by a scalar distance. Rays find extensive use in ray casting operations within spaces and consequently into body geometry (§§ 3.13.2, 3.8.1).

3.8.4.3 Triangles

A triangle in Gravitas is embodied by the structure `Triangle`, characterised by three points representing the vertices. The structure provides methods to compute the area, right-handed normal vector, centroid, edge offset vectors, compute distance and closest point queries, test point containment and ray intersections. Methods are also provided for axial projection, support mappings, and transformations using a **`Transform`** construct (§ 3.8.2).

3.8.4.4 Tetrahedrons

Similarly to triangles, a tetrahedron is represented by the structure **`Tetrahedron`** and is defined by four vertices. Methods are provided for computing volume and centroid, perform distance and closest point queries, perform point containment tests and perform transformations using a **`Transform`** construct (§ 3.8.2).

3.9 Instrumentation Manager

The instrumentation manager module, represented by the class **`InstrumentationManager`**, provides a means for rendering graphical information about the internal state of the physics engine (Figure 65). The manager provides a set of flags to selectively control which types of components are allowed to render instrumentation.

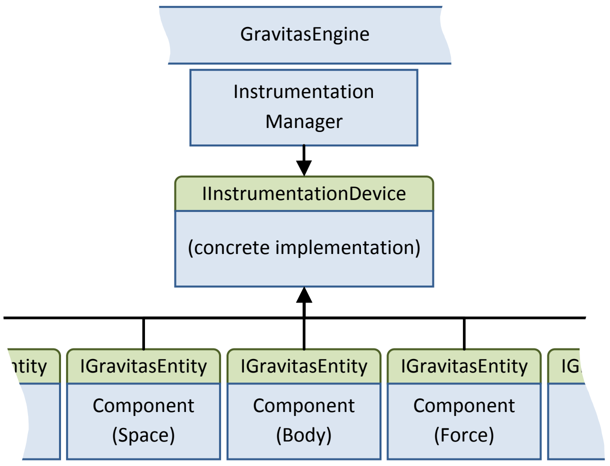


Figure 65 Instrumentation manager

As the physics engine is not bound to any visualisation technology, graphical rendering is accomplished via an abstract graphical device defined by the interface **`IInstrumentationDevice`**. The interface defines a set of elementary graphics services, such as for drawing lines, arrows, boxes and spheres using specific colours and drawing modes.

To render instrumentation, the user is required to implement the device interface using some chosen visualisation technology such as Microsoft Direct X [**73**], or OpenGL [**74**] or a higher level visualisation engine. As a case study, this paper illustrates an implementation (Figure 75) based on the Vistas visualisation engine [**3**].

An implementation for the device in a chosen visualisation technology may be bound to the instrumentation manager that, in turn, may be requested to render a simulation environment's instrumentation according to the enabled flags. The manager achieves this by invoking an instrumentation rendering method for each applicable component. This method is defined in the base Gravitas interface **`IGravitasEntity`** described in (§ 3.3.2). Each component is thus responsible for implementing its own instrumentation graphics. Typical examples include the depiction of forces as directed arrows, bounding volume outlines, spatial structures *etc.* (Figure 76).

3.10 Kinetics Manager

The kinetics manager (Figure 66), represented by class **`KineticsManager`**, is fundamentally a registry for kinetic integrator factories that implement an interface **`IKineticIntegrator`** according to the Abstract Factory pattern [**72**]. The kinetic integrator implementations may be provided by the API user through application code, or may be loaded from third-party plug-in modules.

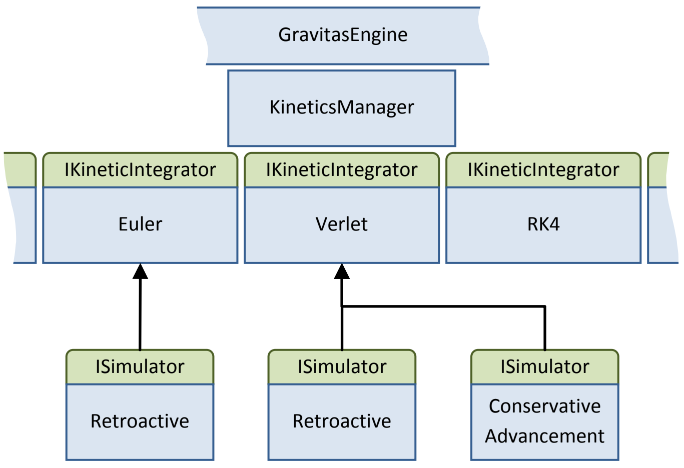


Figure 66 Kinetics manager

3.10.1 Kinetic Integrators

The kinetic integrator interface **`IKineticIntegrator`** is another application of the Strategy pattern [**71**] that abstracts the numerical integration technique applied on a body's kinetic properties. Implementations for the integrator are responsible for integrating motion forward in time for a small time interval. Typical examples include the Euler method, the basic Verlet method, and the RK4 method (§§ 2.4.1 – 2.4.3). The numerical integrator also takes into account the damping coefficient included in the kinetic properties of a body. Rather than opting for a generic numerical integration interface, independent of its application, we chose to define a unified interface to handle both position and velocity integration. The rationale for this decision is that when integrating position and orientation, both the velocity and acceleration terms are exposed to the interface, allowing for second order methods, such as Verlet and RK4, to be implemented.

3.10.2 Kinetic Properties

The structure `KineticProperties` represents a body's position, orientation, linear velocity and angular velocity in world space. The position and orientation are given respectively as a 3D point and a unit quaternion while the linear and angular velocities are given in vector form (§§ 2.3.1.2, 2.3.2.4). To facilitate computations involving the orientation quaternion, its conjugate (A.8) is also stored in the structure.

A simple fluid drag model is also included in the structure, consisting of a velocity damping coefficient as described in (§ 2.6.4).

The kinetic properties structure also stores a timestamp representing the last recorded active state. Coupled with a method that computes an activity or energy level, and a Boolean flag related to activity control, this information is used by the activity manager module (§ 3.10.4) to regulate the activity of a body for reasons of performance.

As collision detection, resolution and constraint solving is carried out in relative space (§§ 2.5 – 2.7), the kinetic properties structure also provides methods for applying a change of basis to a frame of reference dictated by the positional properties of another kinetic properties structure.

3.10.3 Vector Time Functors

A C++ functor is essentially a class with the parenthesis operator overridden such that it may be used as a context-sensitive function. The Gravitas API defines a functor class `VectorTimeFunction` with a time domain and 3D vector range. The vector-time functor class is used to specify time-dependent variables such as forces, torques and linear and angular accelerations. It finds application in force accumulators and kinetic integrators (§§ 3.7.2, 3.10.1).

3.10.4 Activity Management

The activity management module, represented by class `ActivityManager`, is responsible for the sleep/wake functionality of the physics engine. The manager provides methods to test if kinetic conditions are suitable to switching a body to a dormant state, and conversely, for awakening it when it comes into contact with another body or a new force is applied to it. In addition, it also provides methods to propagate activity throughout a network of interacting bodies. This module is a crucial component for ensuring real-time performance for a complex environment where bodies are assumed to settle in static configurations.

3.11 Material Manager

The Material manager module, represented by the class `MaterialManager`, is effectively a central registry for materials describing the surface characteristics of bodies during collisions and contact (Figure 67).

3.11.1 Materials

A material is represented by an instance of the class `Material` that contains a coefficient of restitution property as discussed in (§ 2.6.1.1), static and dynamic coefficients of friction as per (§§ 2.6.1.2, 2.6.2.1, 2.6.3.2) and a surface perturbation coefficient as described in (§ 2.6.5). A material instance is a shared resource and hence, multiple bodies sharing the same surface characteristics may refer to a common material.

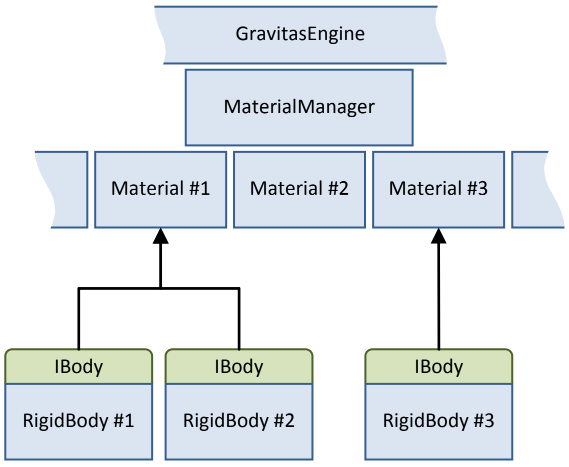


Figure 67 Material manager

3.11.2 Material Mixing

Technically, material properties are empirically derived from experiments involving different material combinations. For instance, a coefficient of restitution may be measured for wood colliding with metal. In Gravitas, these properties are held per material and hence, the engine assumes a technique for deriving properties for mixed material contact and collision.

Given two materials with a high and low restitution coefficients, such as rubber and concrete, the coefficient for rubber-concrete collisions intuitively lies somewhere between the coefficients for rubber-with-rubber and concrete-with-concrete collisions. Hence, Gravitas computes an arithmetic mean of the coefficients of restitution of two materials. We observe that this value still remains within a plausible range and does not exceed the range given by the individual coefficients.

Secondly, Gravitas assumes that the overall contact normal perturbation accumulates the respective surface perturbation of the individual materials, and thus, the respective coefficients are summed together such that their combination yields a more perturbed surface normal. We advise care in the choice of perturbation coefficients such that their combinations yield plausible collision and contact behaviour.

The default material mixing computations provided by the Gravitas engine may be overridden by enabling material events through the event manager and subscribing to these events via a material event listener (§ 3.6.5). This gives the API user the opportunity to perform custom material mixing computations through what is essentially a call-back mechanism.

3.12 Simulation Manager

The simulation manager (Figure 68), represented by class **`SimulationManager`**, is a registry for space factories implementing an Abstract Factory pattern [**72**] through the interface **`ISimulatorFactory`** that defines methods for instantiating simulator instances with corresponding interface **`ISimulator`**. Factory implementations may be provided by application code or external plug-ins.

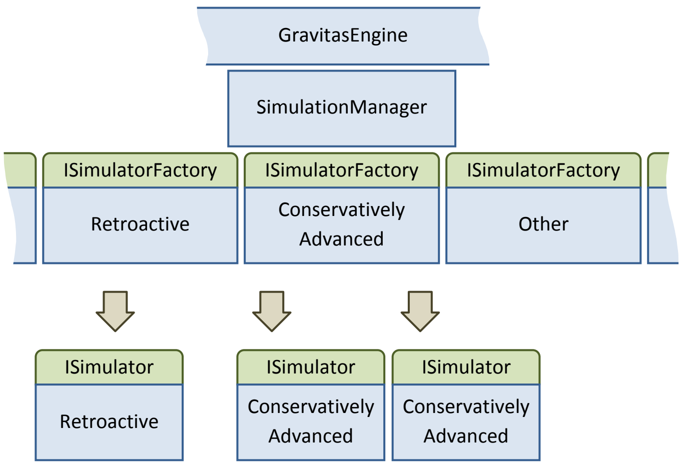


Figure 68 Simulation manager

3.12.1 Simulators

Due to all the possible simulation techniques, the Gravitas physics engine does not impose a fixed simulation pipeline and accordingly, the services provided by the engine are loosely coupled. Thus, the engine allows the API user to define a simulation pipeline through a Strategy pattern [**71**] by implementing the interface **`ISimulator`**. The interface provides methods to initialise and terminate a simulation, and to step through it using a given time delta. On initialisation, the simulator accepts an environment structure (§ 3.12.2) defining the components to be used in the simulation. To date, the engine does not provide automated simulation execution; the API user is required to provide a suitable time-regulated loop in which to step the simulation.

3.12.2 Environments

A Gravitas environment (Figure 69) defines the constituents of a simulation. The environment is represented by the **`Environment`** structure that contains references to a space, a list of constraints, an optional collision filter, a kinetic integrator and a constraint solver (§§ 3.13, 3.5.1, 3.4.3, 3.10.1, 3.5.2). The separation between a simulator and an environment instance provides a means for combining different simulation pipelines with different sub-system implementations. We feel that this approach increases the physics engine's range of applicability.

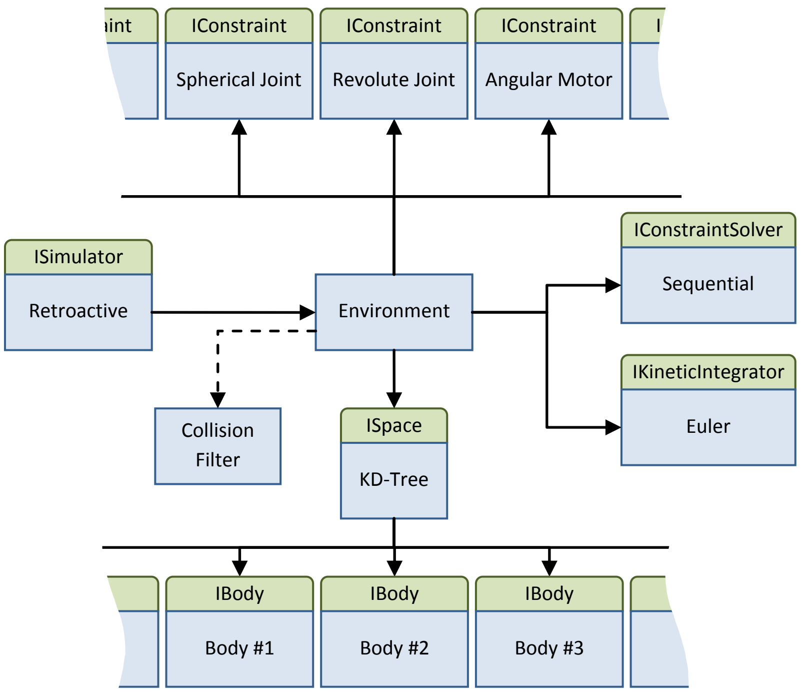


Figure 69 Simulation environment

3.13 Space Manager

The space manager module (Figure 70), represented by class **`SpaceManager`**, is essentially a registry for space factories implemented through application code or external plug-ins. The factories employ the Abstract Factory pattern [72] by implementing the interface **`ISpaceFactory`**. This interface defines methods for creating space implementations, in turn defined by the interface **`ISpace`**.

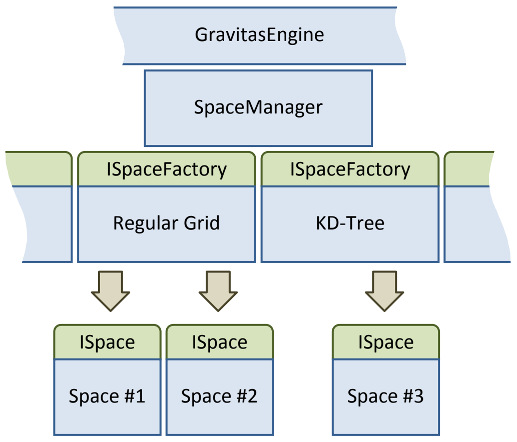


Figure 70 Space manager

An implementation of the space interface is essentially a data structure coupled with a set of algorithms for efficiently managing objects in space. Typical examples include regular grid representations, axis-aligned and arbitrary BSP trees, and coordinate sorting (§ 4.6). The purpose of a space is to facilitate broad-phase collision detection by identifying clusters of potential collisions. As ray casting tests may also benefit from the underlying data structure, we chose to include these tests in the definition of the interface.

The **`ISpace`** interface defines a method for building the underlying representation from scratch. This functionality is intended as an initialisation task or as a relatively infrequent occurrence and hence the space implementation may be allocated a relatively large amount of processing resources to compute an optimal data representation. A typical example is an algorithm for computing an optimal BVH for a space, which may entail using *simulated annealing* [**75**] or some other statistical method.

The space interface also defines a method for updating the spatial representation in order to maintain its validity as the simulation progresses in time. The real-time requirement of this method implies that its implementation is afforded a limited processing budget. Thus, an update may fix a representation at the potential expense of optimality. Quoting the BVH example, the update method may have to deal with a body that is no longer fully enclosed within its node. Instead of restructuring the hierarchy, the method may simply move the body towards the root of the tree until it is fully enclosed in a node, or if no such node is found, the root node is simply extended to enclose the body. Conversely, the body may have attained a placement that allows it to be shifted down to a child node. Using this technique, bodies may move around the hierarchy over time, further reducing its effectiveness. In general, it is sufficient to shift a body one level at a time, allowing it to settle in a suitable node over a number of simulation time steps. The update method may opt to rebuild the structure at occasional intervals or based on some optimality criterion, in order to restore its performance.

3.13.1 Candidate Collision Lists

The primary purpose of a space implementation is to efficiently compute a *candidate collision list*, consisting of pairs of bodies that are in potential collision according to some metric. This is achieved by traversing the underlying data structure of the implementation, using corresponding search algorithms to quickly identify potential collisions. To further speed up computations, a space generally deals with bodies enclosed by simple bounding volumes (§ 3.8.3). This facilitates the construction and update of the spatial representation during the simulation.

3.13.2 Ray Tests

The ray testing functionality provided by a space can often benefit from the underlying spatial representation. For instance, when space is partitioned using a BSP tree, a recursive ray test may be implemented such that the searching algorithm descends only into the partitions intersected by the ray. This results in logarithmic time complexity $O(\log n)$ as opposed to the linear $O(n)$ complexity of a brute-force search.

In general, the search identifies all bodies whose bounding volume is intersected by the ray. For every such body, the ray, represented by the `Ray` structure (§ 3.8.4.2) is transformed using the `Transform` structure (§ 3.8.2) into the local space of the geometry, and is tested against it using a method defined in the `IGeometry` interface (§ 3.8.1).

3.14 Miscellaneous Entities

In this section we discuss physics engine components that do not fall under the direct responsibility of any manager. Nevertheless, these components constitute an important aspect of the engine.

3.14.1 Bodies

In an attempt to generalise bodies, we have specified an interface called `IBody` that amalgamates the characteristics of a body with respect to physical simulation. The interface defines methods to retrieve and set the geometry (§ 3.8.1), the bounding volume (§ 3.8.3), the mass properties (§ 3.14.2), the material (§ 3.11.1), the kinetic properties (§ 3.10.2) and the force accumulator (§ 3.7.2). In addition, a number of methods provide functionality to associate a body with one or more collision groups (§ 3.4.4) that are used in conjunction with the collision filter (§ 3.4.3).

The rigid body implementation is provided by the class `RigidBody` implementing the `IBody` interface.

3.14.2 Mass Properties

The mass and inertia tensor matrix of a body are collectively held in the structure `MassProperties`. To facilitate computations, the mass inverse is pre-computed and stored in as an additional property. Similarly, the tensor matrix inverse is also pre-computed in both the local and the world frame of reference. The resulting matrices are also stored in the structure. The mass property inverses provide a convenient way for defining immovable bodies by setting their values to zero and the zero matrix respectively, as described in (§ 2.6.2.1). To this effect, the `MassProperties` structure provides a method for making the mass immovable. For consistency, the normal mass and tensor properties are assigned the largest positive value allowed in the floating point range. Another method computes a mass ratio with respect to the properties held in the structure and another mass properties structure. A typical application of this ratio is for resolving body interpenetration using linear projection.

Chapter 4
Plug-in and Application Case Studies

In this chapter we present a number of plug-in implementations that substantiate the Gravitas physics engine framework discussed in Chapter 3. The implementation of each plug-in is a case study that applies the techniques and theory surveyed in Chapter 1 and Chapter 2. For this thesis we endeavoured to develop a comprehensive set of core plug-ins to cover all aspects of the physics engine. Subject to available time, we have also provided different implementations for specific sub-systems to illustrate differences in processing and memory footprints as applicable.

Finally, we conclude this chapter by presenting the Gravitas Sandbox application, a tool that enables users to construct and interact with physical environments.

4.1 Core Constraints and Constraint Solvers Plug-in

The Core Constraints plug-in provides a number of implementations for the `IConstraint` and `IConstraintSolver` interfaces. (§§ 3.5.1, 3.5.2).

4.1.1 Constraint Implementations

The constraint implementations include the classes `SphericalJointConstraint`, `RevoluteJointConstraint`, `RigidJointConstraint` and `AngularMotorConstraint` representing ball-and-socket joints, hinges, rigid connections and angular motors as per the proposed models in (§§ 2.7.5.2, 2.7.5.3, 2.7.5.4, 2.7.5.5). In addition, ball-and-socket and hinge implementations specify joint limits (§2.7.5.6) and all joints may be configured to be breakable (§2.7.5.7). Binding parameters, joint limits and other such joint properties are specified via the custom property mechanism.

4.1.2 Constraint Solver Implementations

The constraint solvers implemented by the plug-in include a sequential solver represented by class `SequentialConstraintSolver` and a penalty-based solver represented by `PenaltyBasedConstraintSolver`. The solvers are based on the approaches discussed in (§§ 2.6.3, 2.7.6). Due to limited time, we were unable to refine the penalty-based solver to a stable state, and more work is required in the area of penalty parameter computation as treated by [**49**] and [**50**]. We defer this task as potential future work.

4.2 Core Forces Plug-in

The Core Forces plug-in contains a number of `IForce` interface (§ 3.7.1) implementations as case-studies for different types of forces. On loading into the physics engine, the force implementations are registered by the plug-in with the force manager (§ 3.7) for use in simulations.

The `BasicForce` implementation is useful for creating simple forces defined in terms of a fixed point of application and a fixed force value. A basic force may also be configured as instantaneous if required.

The `GravityForce` implementation is, as its name implies, a representation of the force of gravity, modelled as a constant field of force. The gravity force is parameterised via an acceleration vector due to gravity, specified through the custom property system. When the force implementation is applied to the body, it implicitly assumes the centroid as the point of application and computes the force value as the scalar product of the body mass with the acceleration due to gravity.

The `VortexForce` implementation is a more specialist case-study of the `IForce` interface that simulates a columnar force field. This implementation may be used to simulate objects caught in a tornado or some other vortex-like force field (Figure 75).

Due to limited research time, we have been unable to develop more elaborate case-studies such as geometry-based aerodynamic friction models. We defer such effort as potential future work.

4.3 Core Geometry and Collision Detectors Plug-in

We chose to implement a Core Geometry plug-in module that incorporates both geometry (§ 3.8.1) and the associated collision detector (§ 3.4.1) implementations. The motivation for this decision is to leverage the compile-time information of the implemented code, rather than relying on metadata queries to determine geometric properties during collision detection.

4.3.1 Geometry Implementations

The geometry implementations, extending the `IGeometry` interface, follow a similar approach, each exposing their properties both through the custom property system and as public variables to speed up property access by the collision detector implementations. The volume and the distribution tensor of each type of geometry are computed from established analytical or numerical results (§ 2.3.2.3). Ray tests are also treated on a case-by-case basis, often solved in terms of simultaneous involving the Cartesian or parametric forms of the geometry surface and the ray equations. Arbitrary complex geometry such as convex polyhedra or triangle meshes uses a combination of iterative and analytic approaches.

4.3.1.1 Spheres

Spherical geometry is supported via the `Sphere` class that is defined in terms of a radius and is implicitly assumed to have its centroid coinciding with the local origin.

4.3.1.2 Boxes

Cuboid geometry is supported by the `Box` class, defined in terms of three half extents, with its centroid assumed to coincide with the local origin and its alignment coincident with the local axes.

4.3.1.3 Cylinders

Cylindrical geometry is supported by the `Cylinder` class, defined in terms of a cap radius and longitudinal height, with its centroid assumed to coincide with the local origin and its central axis coincident with the y-axis of the local frame of reference.

4.3.1.4 Convex Polyhedra

Convex polyhedral geometry is handled by the `ConvexPolyhedron` class, defined in terms of a vertex list, and by lists of edges and faces that index into their adjacent vertices according to their position in the list. Although this definition does not implicitly enforce alignment with a local frame of reference, we recommend specifying polyhedra such that their centroids coincide with the local origin and such that they are symmetrical with as many axes as possible.

Convex polyhedra are provided as distinct implementations from other types of polyhedra as collision detection for such geometry can leverage the properties their convex nature, such as the ability to find axes of separation, quick location of support function maxima etc. The volume and distribution tensor properties are computed using iterative techniques [**30**]. Ray testing is performed iteratively by testing against each face.

4.3.1.5 Triangle Meshes

Triangle meshes, implemented by the geometry class `TriangleMesh`, are essentially arbitrary arrangements of triangular faces that may or may not form a closed polyhedron. Such meshes are often used to define complex polyhedral environments potentially consisting of thousands of polygons.

To speed up spatial queries against such geometry, the constituent triangles are arranged in BVH’s (Figure 76), reducing the search algorithm complexity from polynomial to logarithmic time. The BVH algorithm employed for this geometry recursively divides the triangle set by a plane orthogonal to the principal axis yielding the longest span. The triangles are partitioned into two sets according to whether their centroid lies on one side of the plane or the other. The sets are then enclosed in two potentially overlapping bounding volumes, and the process is repeated until a maximum tree depth or minimum node size is achieved. These termination criteria are exposed as custom properties.

The triangle mesh is parameterised either through a list of vertices and faces or by a reference to an external model file in Wavefront OBJ format [**76**]. We chose this format due to its simple, open, text-based specification and wide availability as an export format in many modelling packages. The OBJ model loader used by triangle meshes is a custom loader that extracts only vertex and triangle information, ignoring other elements irrelevant to physics geometry such as groups, materials, normals and texture coordinates. In addition, it automatically triangulates faces of four or more sides to fit the triangle mesh structure.

As a triangle mesh is a potentially discontinuous boundary representation, with no guaranteed notion of volume, we chose to implement mass properties as discrete point mass summations with the elements corresponding to the individual triangles. Each triangle is replaced by a point mass positioned at the triangle centroid and unit mass defined in terms of its area.

4.3.2 Collision Detector Implementations

Implementations of the `ICollisionDetector` interface are provided in the Core Geometry plug-in corresponding to almost all pair-wise combinations of the geometry implemented in the plug-in.

Due to limited time we have omitted a collision detector implementation for testing triangle meshes together. Robust collision detection between such geometry is complicated by the fact that it is difficult to quantify penetration depth, and indeed qualify internal and external regions of a set of triangles with no imposed topological structure. In carefully controlled simulation conditions, we suggest that an implementation based purely on pair-wise triangle contact is possible. Otherwise, the simulation is likely to exhibit tunnelling, visible penetration and implausible collision response. We have also left unimplemented most of the impact time estimation tests due to their complexity. We defer these issues as potential future work.

We have further opted not to implement collision detection between two half-spaces. The reason for this decision is that a half-space is unbounded geometry that is typically used to simulate an immovable body like the ground. Secondly, being unbounded, half-spaces trivially intersect in all cases except when their normals are parallel but facing opposite directions and where any point between the plane boundaries is outside both half-spaces.

4.3.2.1 Sphere with Sphere

The sphere-to-sphere tests of intersection, TOI and generation of contact manifolds are implemented by class `SphereCollisionDetector`, using the analytical approaches discussed in (§§ 2.5.1, 2.5.2.1, 2.5.3.2).

4.3.2.2 Box with Sphere

The class `BoxToSphereCollisionDetector` provides an implementation for intersection testing and contact manifold generation between boxes and spheres. TOI estimation is not implemented to date. The intersection test is implemented according to the box-to-sphere example in (§ 2.5.1) based on a closest point query. The contact manifold is computed as a single contact point positioned at the closest point within the box to the sphere centre and the penetration is computed as the difference of the sphere radius and the closest point distance. The collision detector operates in the frame of reference of the box so simplify computations. The TOI estimation query is not implemented to date.

4.3.2.3 Box with Box

The box-to-box implementation **`BoxCollisionDetector`** uses the SAT method (§ 2.5.1.1) to determine intersection. For contact manifold generation, it implements a variant of this method to determine the axis of least overlap. A support function is then applied along this axis to identify the features in contact. As boxes have parallel sides, the axes tested consist of the three principal axes of each box, and the nine normals resulting from the vector product of the pair-wise edges, parallel to the principal box axes, for a total of fifteen test axes. No implementation for TOI estimation is provided to date.

4.3.2.4 Cylinder with Sphere

The cylinder-to-sphere implementation **`CylinderToSphereCollisionDetector`** essentially projects the intersection testing and manifold generation problem in two dimensions (2D), the plane being defined by the cylinder axis and sphere centre contained within it. This reduces the problem to a 2D box being tested against a circle, and a similar approach to (§ 4.3.2.3) may be used. The resulting 2D contact is then projected to back to 3D to yield the desired contact point. No implementation for TOI estimation is provided to date.

4.3.2.5 Cylinder with Box

Cylinders are tested against boxes by the **`BoxToCylinderCollisionDetector`** implementation. The intersection test is based on SAT (§ 2.5.1.1) where the candidate axes tested are the principal axes of the box, the cylinder axis, and its vector product with the box axes. The axis of least overlap provides a means for computing a contact manifold. In the special case where a cylinder cap is in flat contact with a side of the box, a number of points are sampled along the circular cap edge. No implementation for TOI estimation is provided to date.

4.3.2.6 Cylinder with Cylinder

Cylinders are tested against each other by the **`CylinderCollisionDetector`** implementation. The intersection test is based on the SAT method where the candidate axes tested are the cylinder axes and a third axis along the closest points between the cylinder *shafts*, each consisting of the line segment spanning the cylinder cap centres. We chose this computation for the third test axis over a simple vector product of the first two axes, as the latter approach fails when the cylinder axes are coplanar. No implementation for TOI estimation is provided to date.

4.3.2.7 Convex Polyhedron with Sphere

The `ConvexPolyhedronToSphereCollisionDetector` uses a variation of the GJK algorithm (§ 2.5.1.2) to compute the closest internal point to the sphere centre, and hence test the resulting distance against the sphere radius. The contact manifold is generated using the closest point as the contact point with a penetration depth consisting of the radius to closest distance difference. No implementation for TOI estimation is provided to date.

4.3.2.8 Convex Polyhedron with Box

The `ConvexPolyhedronToBoxCollisionDetector` uses SAT and its variant giving the axis of least overlap to determine intersection and compute a manifold respectively. The candidate axes consist of all the polyhedron face normals, the principal box axes and the vector products of their pair-wise combinations. For a polygon of n faces, this gives a total of $4n + 3$ candidate axes. Each axis test entails the projection of both polyhedron and box on to the axis. Hence, the polyhedron faces whose normals face away from the box centroid are culled to reduce the total number of axes tested. No implementation for TOI estimation is provided to date.

4.3.2.9 Convex Polyhedron with Cylinder

The `ConvexPolyhedronToCylinderCollisionDetector` implementation takes an approach similar to (§ 4.3.2.8). The candidate axis consist of all the face normals facing towards the cylinder, the cylinder axis, an axis along the line joining closest point on the cylinder shaft to the polyhedron centroid, and the vector products of all the applicable face normals with the cylinder axis. No implementation for TOI estimation is provided to date.

4.3.2.10 Convex Polyhedron with Convex Polyhedron

The `ConvexPolyhedronCollisionDetector` implementation also uses SAT and the axis and least overlap where the candidate axes consist of the face normals of both polyhedra involved in the test, excluding the normals directed away from the intersection. The remaining test axes are computed from the edge pair-wise vector products to handle cases where polyhedra come into edge-wise contact. No implementation for TOI estimation is provided to date.

4.3.2.11 Triangle Mesh with Sphere

The `TriangleMeshToSphereCollisionDetector` implementation uses a recursive algorithm to descend into the triangle BVH taking all paths where the bounding boxes of the respective nodes intersects the sphere. For every such node, the contained triangles are individually tested against the sphere. In the case of intersection testing, the algorithm terminates on the first positive result, or otherwise terminates when the search is exhausted. For contact manifold generation, the BVH is exhaustively traversed, given the bounding volume intersection conditions and all intersecting triangles are involved in the manifold. The implementation supports complex meshes consisting of thousands of triangles in real-time performance. The triangle with sphere test is based on a closest point query and distance to radius comparison. No implementation for TOI estimation is provided to date.

4.3.2.12 Triangle Mesh with Box

The `TriangleMeshToBoxCollisionDetector` implementation adopts the same approach as (§ 4.3.2.12), replacing the triangle-sphere tests with triangle-box tests. These elemental tests are implemented with SAT and least axial overlap, using the box principal axes and triangle normal. No implementation for TOI estimation is provided to date.

4.3.2.13 Triangle Mesh with Cylinder

The `TriangleMeshToCylinderCollisionDetector` implementation adopts the same approach as (§ 4.3.2.12), replacing the triangle-sphere tests with triangle-cylinder tests. These elemental tests are implemented with SAT and least axial overlap, using the triangle normal and an axis along the line joining the closest point on the cylinder shaft. No implementation for TOI estimation is provided to date.

4.3.2.14 Triangle Mesh with Convex Polyhedron

The `TriangleMeshToConvexPolyhedronCollisionDetector` implementation adopts the same approach as (§ 4.3.2.12), replacing the triangle-sphere tests with triangle-convex polyhedron tests. These tests use SAT and least axial overlap, where the candidate axes consist of the suitably directed face normals, the triangle normals, and the edge-wise vector product combinations of all the polyhedron edges with the triangle edges. No implementation for TOI estimation is provided to date.

4.3.2.15 Half-Space with Sphere

The `HalfspaceToSphereCollisionDetector` implementation uses the approaches illustrated in (§§ 2.5.1, 2.5.2.2) to implement intersection testing and TOI estimation. The contact manifold is generated by using the closest point on the half-space to the sphere, which trivially evaluates to the sphere centre projected on the xz-plane when testing in the frame of reference of the half-space. The half-space plane normal, pointing along the y-axis is used as the contact normal, while the depth is simply the difference of the sphere radius and the y-component of the sphere centre.

4.3.2.16 Half-Space with Box

The `HalfspaceToBoxCollisionDetector` implementation assumes operation in the half-space's frame of reference and trivially determines intersection with the box by simply checking if any of the box's vertices have a negative y-component indicating penetration. The contact manifold is generated using a similar approach as illustrated in (§ 2.5.3.3). No implementation for TOI estimation is provided to date.

4.3.2.17 Half-Space with Cylinder

The `HalfspaceToCylinderCollisionDetector` implementation assumes operation in the half-space's frame of reference and projects the cylinder onto the y-axis. Intersection occurs if the resulting interval contains zero. In this case, the contact manifold is generated by applying a support function on to the cylinder along the y-axis. For stability, multiple points are sampled when the cylinder lies with its shaft parallel or perpendicular to the half-space's plane boundary. These samples may consist of points along one of the circular cap edges, or along a line perpendicular to both circular cap edges such that it lies on the curved cylinder surface. No implementation for TOI estimation is provided to date.

4.3.2.18 Half-Space with Convex Polyhedron

The `HalfspaceToConvexPolyhedronCollisionDetector` implementation assumes operation in the half-space's frame of reference and trivially determines intersection with the polyhedron by simply checking if any of its vertices have a negative y-component indicating penetration. The contact manifold is generated using a similar approach. No implementation for TOI estimation is provided to date.

4.3.2.19 Half-Space with Triangle Mesh

The `TriangleMeshToHalfspaceCollisionDetector` implementation adopts the same approach as (§ 4.3.2.12), replacing the triangle-sphere tests with triangle to half-space tests, where the latter is transformed into the frame of reference of the mesh. These elemental tests are implemented by computed the signed distance of the triangle vertices from the half-space plane boundary. Negative distances indicate contact and their magnitude gives the penetration. The transformed half-space plane boundary normal is used as the contact normal. If the meshes are very complex, they may contain many coplanar triangles in contact with the half-space leading to an excessively complex contact manifold. Thus, if the manifold contains more than eight contacts it is reduced by random sampling.

4.4 Core Integrators Plug-in

The Core Integrators plug-in contains the three kinetic integrator (§ 3.10.1) implementations `EulerKineticItnegrator`, `VerletKineticItnegrator` and `RK4KineticItnegrator`, respectively implementing the Euler, Verlet and RK4 numerical integration methods (§§ 2.4.1 – 2.4.3) for the linear and angular properties of motion. When the plug-in is loading within the physics engine, it registers the kinetic integrators with the kinetics manager (§ 3.10) for use in physics simulations.

For most simulations, the Euler-based integrator is more than adequate. However, for better accuracy particularly involving highly variable forces, such as stiff springs, the Verlet-based and RK4-based integrators may more suitable options despite their slightly higher computational requirements.

4.5 Core Simulators Plug-in

The Core Simulators plug-in is intended as a provider for a number of implementations of the **`ISimulator`** interface, reflecting different simulation approaches.

The implementation **`BasicSimulator`** is essentially a simple retroactive simulation pipeline where every time-step consists in integrating motion forward in time, updating activity states, perform collision detection and constraint solving, and issue the appropriate event notifications.

To date, we have been unable to implement more advanced simulation pipelines, such as those involving conservative advancement [**15**], shock propagation [**20**] and continuous collision detection [**14**] due to time constraints. We defer this effort as potential future work.

4.6 Core Spaces Plug-in

The Core Spaces plug-in provides a number of implementations for the **`ISpace`** interface, using different spatial representations as surveyed in Chapter 1

The class **`BasicSpace`** is a very rudimentary implementation that computes candidate collision pairs using a brute-force combinatorial approach. Similarly, ray tests are performed by exhaustive testing of all the contained bodies. We recommend usage of this implementation only for the simplest of configurations involving few bodies, or as a robust, albeit inefficient benchmark against which to test more advanced implementations.

The class **`GridSpace`** is an implementation based on the regular grid concept. The grid is assumed of a sparse nature and is hence implemented using a hash-based map with integral cell coordinates as keys. The grid-based space is parameterised by a cell size and a cleanup interval through the custom property mechanism. The cleanup interval parameter defines a regular time interval controlling the rate at which empty cells are removed from the map. The candidate collision list computation benefits from the grid representation, however, ray tests are implemented in the same manner as in the **`BasicSpace`** class as the grid does not reduce the number of bodies tested.

The class **`KDTreeSpace`** is an implementation based on axis-aligned BSP trees, commonly known as *KD-trees*. The representation building method attempts to compute an optimal tree structure by recursively partitioning bodies along the current largest axial span. Custom properties control the maximum depth and the minimum bodies allowed in a node. Dynamic updates are performed by simply migrating bodies upwards or downwards in the tree such that each body is wholly inside a partition. The dynamic update occasionally rebuilds the KD-tree to restore its balance. This implementation provides candidate collision list generation and ray testing that scales very well for large environments due to the logarithmic search algorithm complexity characteristic of the KD-tree structure.

The class **`SweepAndPruneSpace`** is an implementation based on coordinate sorting along one or more axes [**77**]. The algorithm for candidate collision list generation operates in linear time as the sorted coordinate lists are traversed. Ray testing does not benefit however, and is implemented as in the **`BasicSpace`** class.

4.7 Gravitas Sandbox Application

We developed the Gravitas Sandbox application (Figure 71) as a means for quickly constructing physics simulations powered by the Gravitas physics engine. The simulation environments are defined using a text-based format, a sample of which is provided in Appendix B. The sandbox application graphics are powered by the Vistas visualisation engine [**3**] using the application framework provided by Vistas.

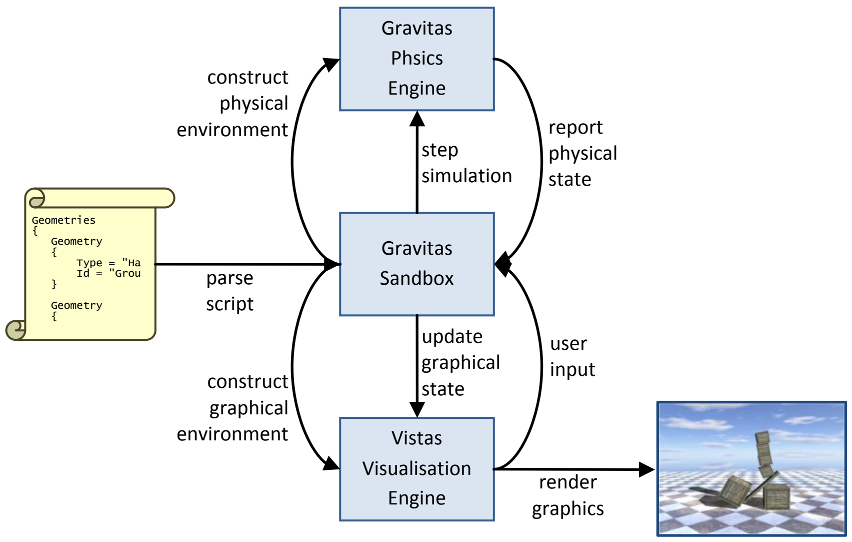

Gravitas Phsics Engine
construct physical environment
step simulation
report physical state
Geometries { Geometry { Type = "Ha Id = "Grou } Geometry {
parse script
Gravitas Sandbox
update graphical state
user input
construct graphical environment
Vistas Visualisation Engine
render graphics

Figure 71 Modular organisation of the Gravitas Sandbox Application

On execution, the sandbox application reads simulation script files provided from the command-line, builds up the corresponding physical environment, and renders the simulation in its initial configuration. Through the functionality provided by the Vistas application framework, the user is able to pan and move around the camera using a combination of keys and the mouse. The simulation may be activated, paused or reset using a number of function keys. Additionally, the user is given the opportunity to interact with movable bodies by dragging then with the mouse using the right mouse button. This is achieved by creating a spring force linking the body to the mouse position projected spherically into the world space. The spring force is further modulated by the mass of the body to achieve similar responses regardless of body size. To aid debugging, the application allows the user to enable or disable rendering of instrumentation, and to lock the camera on a specific body.

Chapter 5
Evaluation

We feel that we have achieved, to a major extent, the goal for a flexible physics engine framework supporting the implementation of a multitude of simulation techniques, entities and algorithms to cooperate within an environment at real-time animation rates. The degree to which this goal has been achieved is reflected in the number of alternative implementations that were, or may potentially be developed, for every abstracted aspect of the engine. Through the core plug-ins we have provided multiple implementations for geometry, constraints, spatial representations and forces. However, to date we have only provided single implementations of other aspects, that we hence suggest as potential areas of further work.

We have also succeeded to a reasonable extent in constructing an abstract specification for physical constraints, providing mathematical models and concrete implementations to substantiate the specification.

Another measure of the success, or otherwise, of this work is the perceived realism of the exemplary simulations constructed throughout the study. Regrettably, we are unable to define, for arbitrarily complex configurations, an objective assessment of the simulation quality achieved by the Gravitas physics engine framework. We must therefore rely on our natural sense of physical plausibility and of that of the audience observing the animations generated by the engine.

In this chapter we present a number of screen captures illustrating simulations generated with the Gravitas Sandbox application. These simulations are intended to cover the full feature set of the physics engine framework and the associated plug-in modules developed to support it.

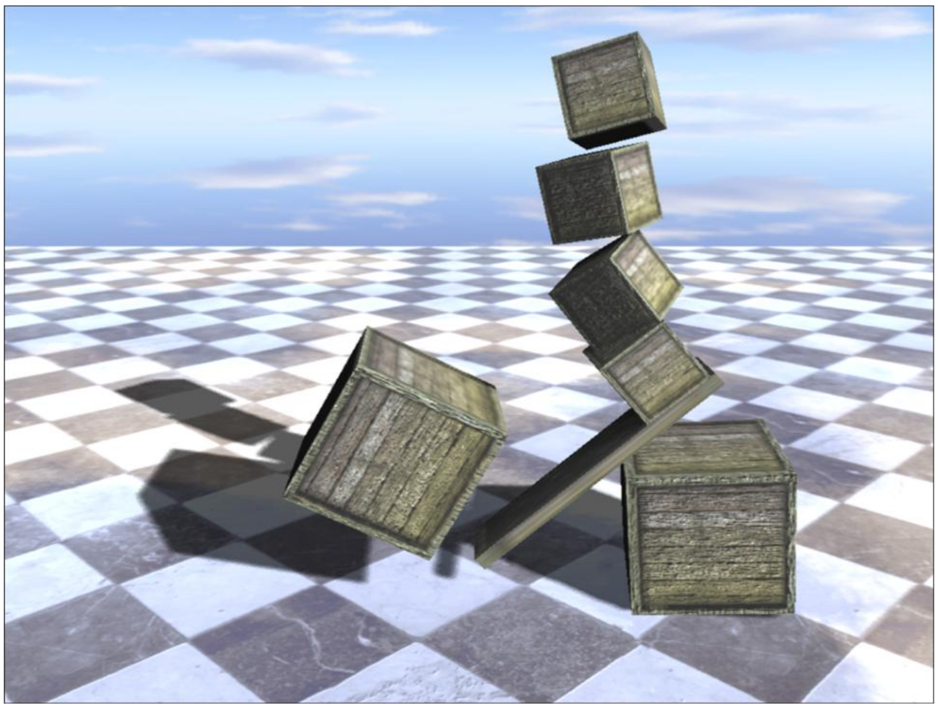


Figure 72 A stacked configuration of boxes leveraged by a counter weight

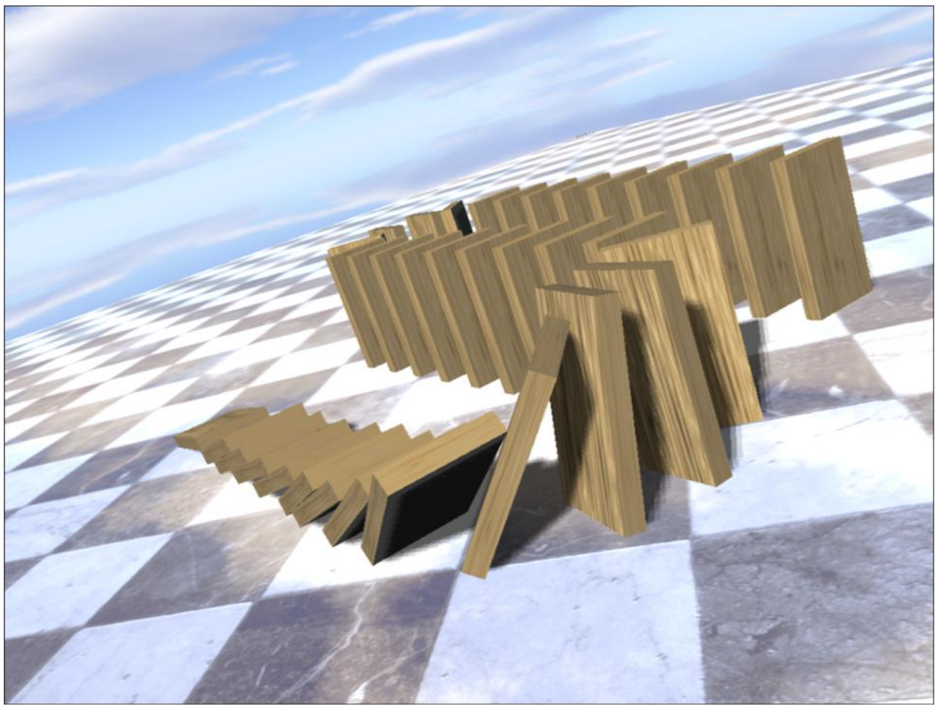


Figure 73 A simulation of the domino effect

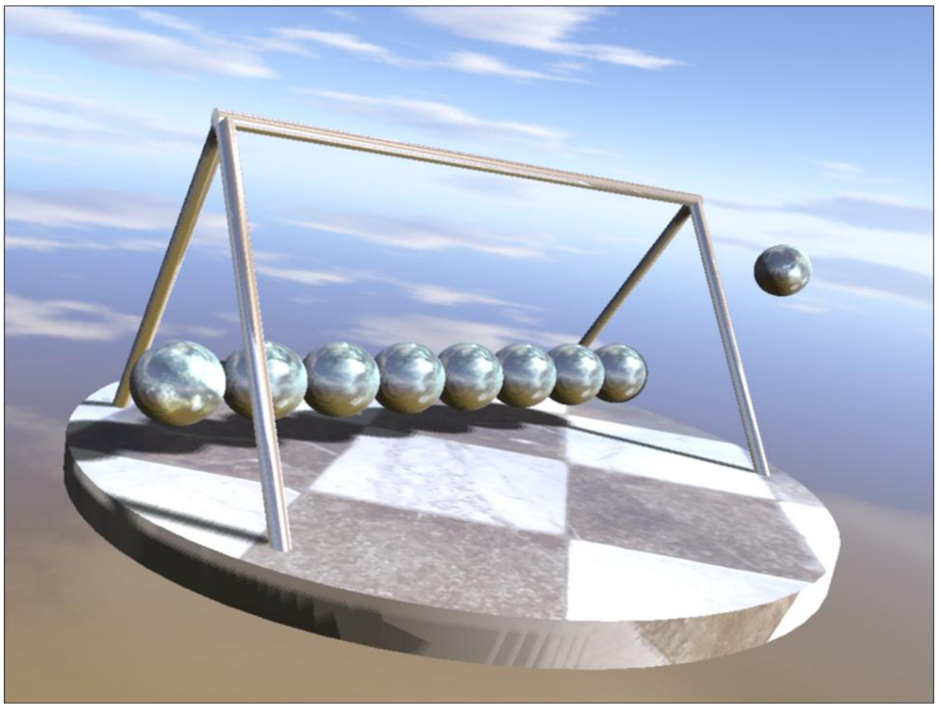


Figure 74 A demonstration of Newton’s Cradle

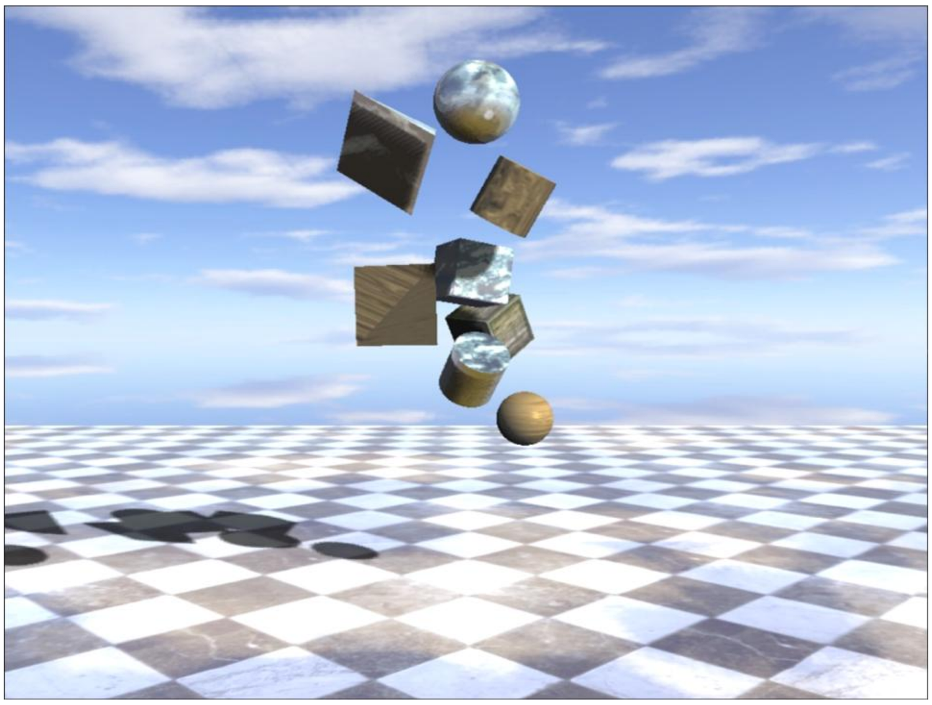


Figure 75 Force implementation example illustrating bodies caught in a simulated vortex

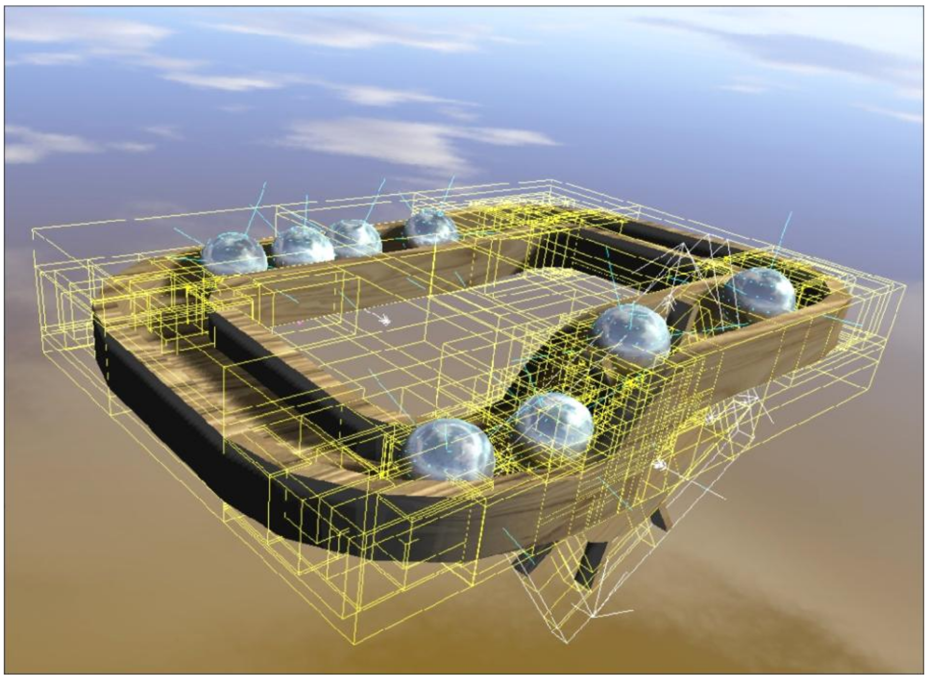


Figure 76 Simulation with instrumentation enabled

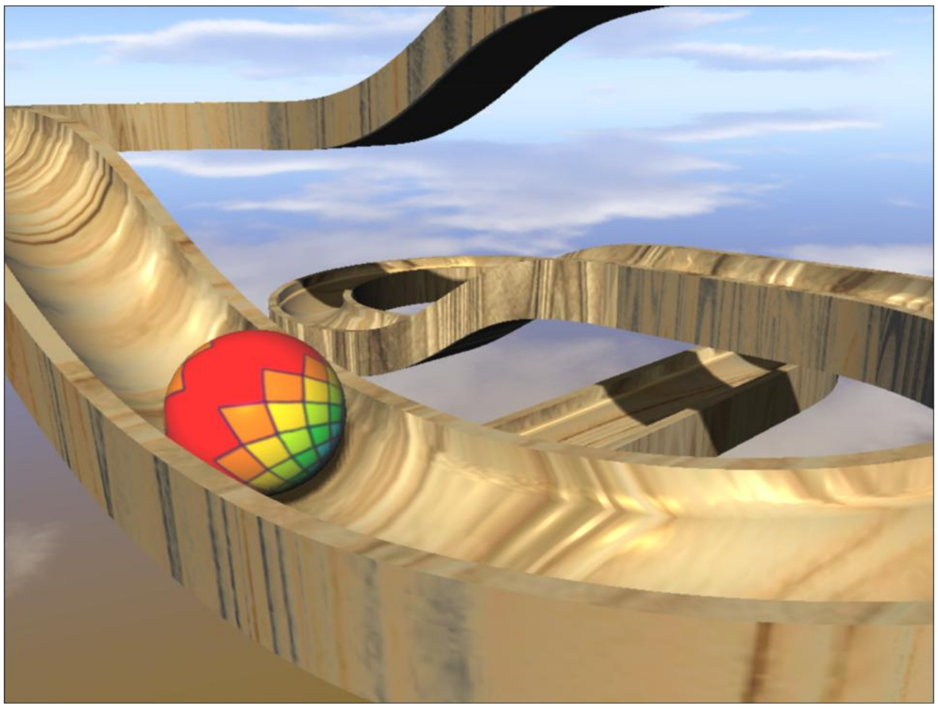


Figure 77 A ball running down a triangle-mesh track (17784 vertices, 35564 faces)

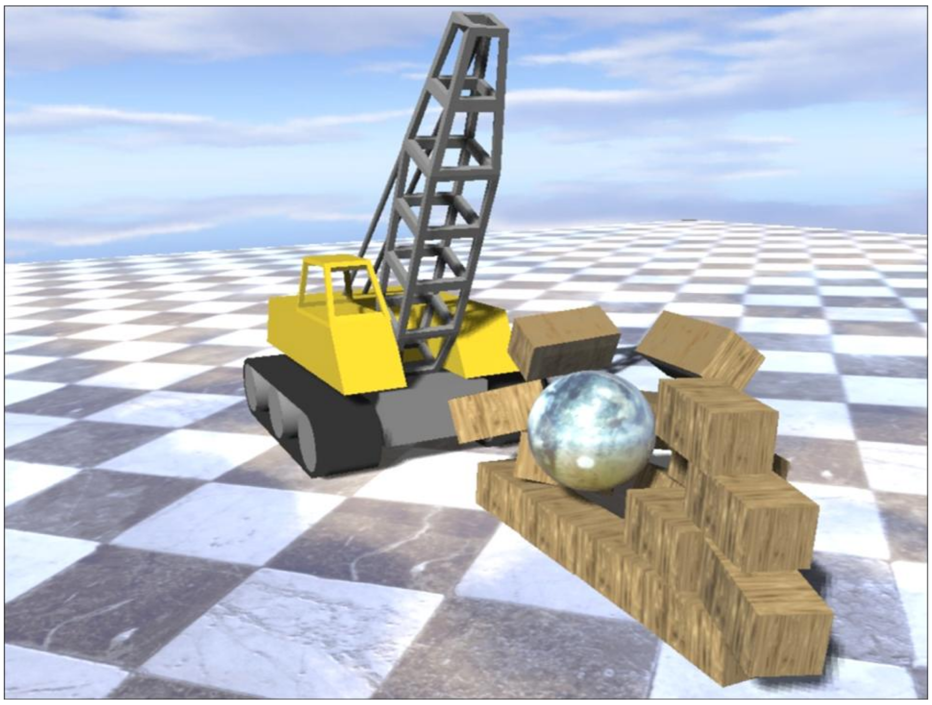


Figure 78 A simulation of stacked and articulated bodies

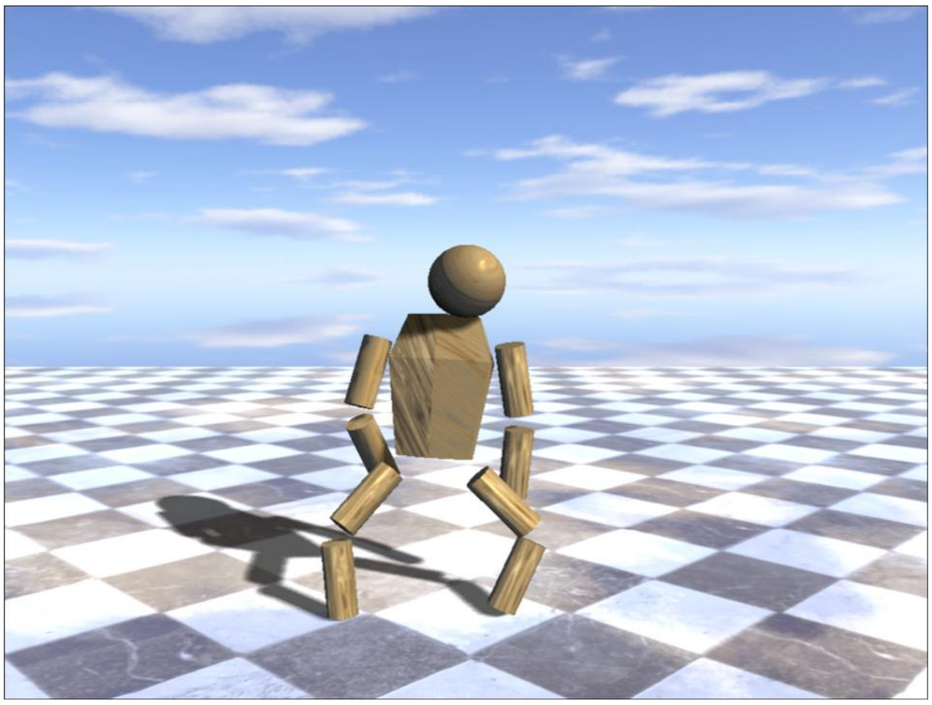


Figure 79 Ragdoll physics using spherical and revolute joints

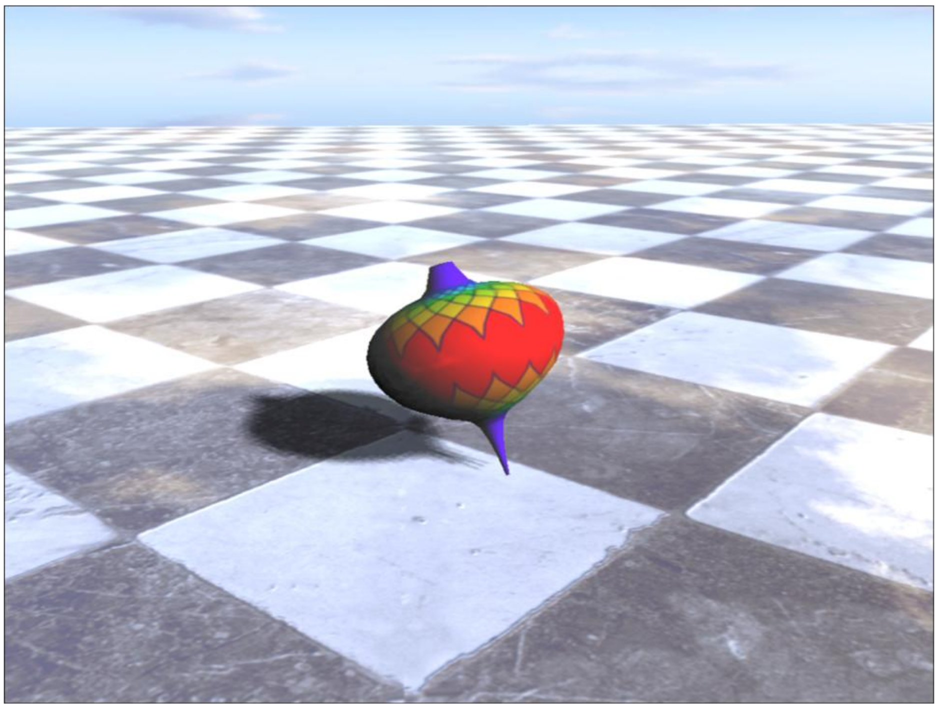


Figure 80 A spinning top exhibiting precession

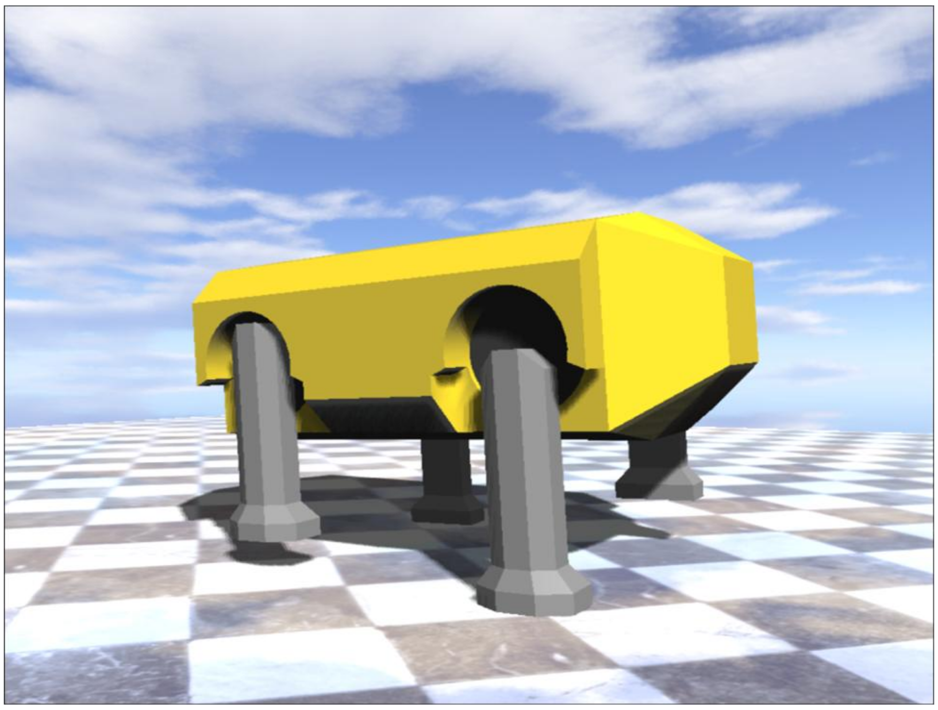


Figure 81 A simulated mechanical quadruped

Chapter 6
Conclusions and Future Work

In this chapter we draw our conclusions with regards to the degree of success in achieving the goals we set out for this dissertation. We also highlight areas for potential future work to further substantiate the results obtained, or enhance upon them.

6.1 Conclusions

Unlike other dissertations that typically focus on a single aspect of physics simulation, this paper is an attempt at unifying a diverse but related set of concepts in the domain of interest, whilst maintaining a pragmatic approach. This has presented us with a number of challenges, principally, the need for reviewing a wide range of techniques and associated theory related to physical simulation in order to identify areas of commonality upon which to build the engine framework. We have demonstrated that it is practical to architect a sufficiently generic physics engine framework operating at real-time rates, accommodating a wide range of simulation techniques and features on commodity computer hardware.

6.2 Future Work

We are convinced that the Gravitas physics engine framework is ripe for extension and enhancement and we hereunder highlight a number of lacunae in this paper that we suggest as amenable to further research or work.

6.2.1 Constraint Model Validation

In (§ 2.7.4) we proposed an abstract constraint model specification, together with a number of exemplary models including non-penetration and a number of common joints. We have constructed these models based on geometrical intuition and observed results. However, we have refrained from proving that these models are indeed what they purport to be, from a mathematical perspective. For instance, we have observed that a hinge joint may be constructed by binding two bodies at two given points passing through the desired axis of revolution, but we have not proved that this indeed produces the required revolute constraint. We hence suggest further work involving derivation of appropriate constraint equations for such joint constructs, for instance, comparable to the Cartesian constraints from which Jacobian formulations are derived.

6.2.2 Constraint Solver Implementations

We have restricted ourselves to implementing only one approach for solving constraints; a sequential solver as described in (§ 2.7.6). To further substantiate the constraint model specification proposed in (§ 2.7.4), we suggest further work involving other approaches such as the simultaneous LCP formulation and a stable penalty-based solver (§§ 2.6.7.2, 2.6.3).

6.2.3 Force Implementations

To date, the force implementation repertoire is very limited (§ 4.2) and does not truly reflect the intended flexibility of force implementations within Gravitas. We hence suggest further work involving the creation of more sophisticated forces, such as buoyancy or aerodynamic drag.

We suggest a buoyancy model based on Archimedes' principle, parameterised by a fluid level or region, and a fluid density factor. The model may be implemented by computing a proportion of the geometry immersed in the fluid and sampling elemental forces at regular intervals within the immersed portion of the geometry. Summing these elements yields a sum force vector value and application point suitable for the force interface. The application point and magnitude of this force is expected to oscillate across the body, preventing it from sinking too much in any one direction, and producing the desired swaying motion.

As a potential approach for fluid drag forces, we speculate on a model based on the relative motion of each face in the geometry of a body, with respect to some fluid field. An elemental force is computed at the face centroid dependent on its area, incidence of motion with respect to the fluid and a viscosity factor. The elemental forces may be then summed up into a single vector value and application point as required by the force interface. If the geometry is such that it cannot be decomposed into faces, a representative face may be assumed. For instance, the area facing fluid drag for a sphere may be approximated by integrating the incidence of the relative fluid motion over the applicable surface area, deriving a value parameterised by the sphere radius and fluid motion, which may, for the sake of simplicity, be assumed constant around the region occupied by the sphere.

6.2.4 Estimated Time of Impact Query Implementations

We have provided implementations for the estimated TOI methods only for the simplest of geometry as detailed in (§ 4.3.2). We hence suggest further work in this area, based on research such as [**39**], [**40**], [**41**] and [**15**]. These implementations enable simulations using continuous collision detection and further substantiate the architecture of the physics engine as a multi-technique environment.

6.2.5 Simulation Pipeline Implementations

We have limited ourselves to a retroactive simulation pipeline but have not provided implementations based on conservative advancement, continuous collision detection and other such advanced methods. We suggest this is a further area of work to validate the abstract simulation pipeline provided by Gravitas.

6.2.6 Robust implementation of Triangle Mesh Collisions

Triangle meshes are very powerful geometrical constructs that allow for simulations involving complex environments that may be modelled using appropriate software packages in a similar manner to visual geometry. However, these meshes are fundamentally surface representations and hence present challenges when tested against other surface-based geometry, namely other triangle meshes. We suggest this as an area of study and direct the interested reader to potential approaches, such as attempting to build depth maps to quantify and qualify internal regions, or to sweep the elemental triangles into prismoids by factoring in time, or their position from an earlier frame, to minimise tunnelling.

6.2.7 Enhanced Sandbox Application

In the time available, we have endeavoured to develop a sandbox application to allow for experimentation with the Gravitas physics engine. This application is based on the Vistas visualisation engine [**3**] but does not really do justice to its graphics capabilities. Further, user interaction is limited to simply providing a means to drag bodies around. Such interaction may be greatly facilitated by the introduction of true scripting functionality within the sandbox, such as the Cuneus scripting engine [**8**].

We, in conjunction with the respective authors of [**3**] and [**8**], therefore suggest the development of a unified sandbox application, termed the Meson Sandbox, providing more control over the visual, physical and scripting elements. For instance, it would be possible to specify complex visual geometry approximated by simpler physical geometry. Additionally, the visual scene graph could be configured in a potentially more sensible manner than the scene graph implicitly assumed by the current sandbox application. Finally, it would be possible to provide custom behaviour and interaction with the environment using scripting, allowing for a wide spectrum of applications, without the requirement of specific application hard-coding.

As a motivating example, we consider a car driving simulation, whereby the car and the surrounding environment are represented by detailed graphical 3D models using the Vistas visualisation engine, car physics and interaction with the environment is provided by Gravitas and user driving controls and independent agents such as other computer-controlled cars are scripted using Cuneus.

6.2.8 Soft Body Dynamics

Our next major goal is to extent the Gravitas physics engine to support soft body dynamics. This may entail refactoring of some elements within the engine framework, to accommodate such bodies and the associated techniques for their simulation. We trust that the design we have architected so far is, to a greater extent, agreeable to soft body support.

Glossary

This section is a glossary of common abbreviations and mathematical notation conventions used throughout this document.

Table of Abbreviations

2D	two-dimensional (or two dimensions)
3D	three-dimensional (or three dimensions)
4D	four-dimensional (or four dimensions)
BSP	binary space partition (tree)
BVH	bounding volume hierarchy
CGI	computer-generated imagery
CM	centre of mass
CPU	central-processing unit
CV	continuous-velocity (joint)
GJK	Gilbert-Johnson-Keerthi (convex polyhedron intersection algorithm)
LCP	linear complementary problem
PSD	positive semi-definite (or positive semi-definitiveness of a) matrix
SAT	separating axis test(s)
TOI	time of impact (estimate)

Mathematical Notation

All mathematical variables are denoted in italics and further differentiated in terms of their dimensions via capitalisation and boldface. Scalar variables are denoted in small lowercase italic letters, such as a, b, and c. We distinguish individual variables by assigning different letters from the Latin or Greek alphabet, or by assigning subscripts to the same letter, such as x_1, x_2, x_3, ... *etc.*

Vectors, tuples and quaternions are denoted in small bold italic letters such as $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$. Matrices and sets are denoted by bold, italicised, capital letters, such as $\boldsymbol{C}$ or $\boldsymbol{D}$.

We classify every new variable introduced in a discourse by specifying the type or set of which it is member, such as $a \in \boldsymbol{A}$. We denote the natural numbers, real numbers and quaternions by the type sets $\mathbb{N}$, $\mathbb{R}$.and $\mathbb{H}$ respectively. We further denote the Boolean set of truth and falsity by $\mathbb{B} = \{T, F\}$. We adopt superscripted indices for vector types, such as $\mathbb{N}^2$ and $\mathbb{R}^n$ respectively for the set of 2D 'vectors' defined over the natural numbers and for the set of n-dimensional real vectors. Similarly, we specify matrix types using index notation such as $\mathbb{R}^{3\times 4}$ denoting the set of all real three row by four column matrices. We remark that $\mathbb{R}^1$ and $\mathbb{R}^{1\times n}$ are written more succinctly as $\mathbb{R}$ and $\mathbb{R}^n$ respectively.

We decompose a vector in terms of its elements using row vector notation, such as $\boldsymbol{a} = [a_1 \quad a_2 \quad a_3]$ where $\boldsymbol{a} \in \mathbb{R}^3$**.** For reasons of convenience, we may alternatively decompose a vector in terms of sub-vectors, such as $\boldsymbol{a} = [\boldsymbol{b} \quad \boldsymbol{c}]$ where, for instance, $\boldsymbol{a} \in \mathbb{R}^5$, $\boldsymbol{b} \in \mathbb{R}^2$ and $\boldsymbol{c} \in \mathbb{R}^3$,. For multiplication with matrices, we denote the transposition of a row vector $\boldsymbol{a} = [a_1 \quad a_2 \quad a_3]$ to the column vector $\boldsymbol{a}^T$, given by

$$\boldsymbol{a}^T = [a_1 \quad a_2 \quad a_3]^T = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Vector equality, denoted by $\boldsymbol{a} = \boldsymbol{b}$, is understood as the element-wise equalities $a_i = b_i$, while vector inequality, denoted by $\boldsymbol{a} < \boldsymbol{b}$ or $\boldsymbol{a} \leq \boldsymbol{b}$, is analogous to the element-wise inequalities $a_i < b_i$ or $a_i \leq b_i$ being all satisfied. A similar interpretation is assumed for $\boldsymbol{a} > \boldsymbol{b}$ and $\boldsymbol{a} \geq \boldsymbol{b}$. We specify vector addition and subtraction using scalar operator symbols, for instance, $\boldsymbol{a} + \boldsymbol{b}$ and $\boldsymbol{a} - \boldsymbol{b}$, where $(-\boldsymbol{b})$ denotes the additive inverse of vector $\boldsymbol{b}$, such that $\boldsymbol{b} + (-\boldsymbol{b}) = \boldsymbol{0}$ is the zero vector. We denote multiplication of a vector $\boldsymbol{a}$ by a scalar $\lambda \in \mathbb{R}$ simply as $\lambda\boldsymbol{a}$ or $\boldsymbol{a}\lambda$. We distinguish between the scalar and vector products using different operators, that is, $\boldsymbol{a} \cdot \boldsymbol{b}$ and $\boldsymbol{a} \times \boldsymbol{b}$ for the respective products. For convenience we denote the scalar product of a vector $\boldsymbol{a}$ with itself, that is $\boldsymbol{a} \cdot \boldsymbol{a}$, as $\boldsymbol{a}^2$. Further, we denote the magnitude of a vector $\boldsymbol{a}$ by $|\boldsymbol{a}|$ and a unit magnitude vector by applying a *circumflex,* such as $\hat{\boldsymbol{a}}$.

We decompose an n-row by m-column matrix in terms of its elements using the notation $\boldsymbol{A} = [a_{ij}]$ for $1 \leq i \leq n$ and $1 \leq j \leq m$, or more explicitly as

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}$$

We alternatively decompose a matrix in terms of sub-matrices for reasons of convenience, such as

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix}$$

where $\boldsymbol{M} \in \mathbb{R}^{4\times4}$, $\boldsymbol{R} \in \mathbb{R}^{3\times3}$, $\boldsymbol{t} \in (\mathbb{R}^3)^T$, $\boldsymbol{0} \in \mathbb{R}^3$ and $1 \in \mathbb{R}$.

We denote equality of two same-size matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ by $\boldsymbol{A} = \boldsymbol{B}$, defined in terms of element-wise equality, that is, $a_{ij} = b_{ij}$ for all valid i and j. We similarly denote inequalities such as $\boldsymbol{A} \leq \boldsymbol{B}$, meaning $a_{ij} \leq b_{ij}$ for all valid i and j. We denote addition and subtraction of matrices of equal dimensions by $\boldsymbol{A} + \boldsymbol{B}$ and $\boldsymbol{A} - \boldsymbol{B}$, defined in terms of their element-wise addition and subtraction $a_{ij} + b_{ij}$ and $a_{ij} - b_{ij}$. We denote multiplication of matrix $\boldsymbol{A} \in \mathbb{R}^{n\times p}$ with matrix $\boldsymbol{B} \in \mathbb{R}^{p\times m}$ as $\boldsymbol{AB}$, such that $\boldsymbol{C} = \boldsymbol{AB}$ and $c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. We denote the zero-matrix by $\boldsymbol{0}$ where its dimensions are assumed from the context. We similarly denote the identity matrix $\boldsymbol{1}$ such that $\boldsymbol{A1} = \boldsymbol{1A} = \boldsymbol{A}$ for all square matrices $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ where the identity matrix is given by

$$\boldsymbol{1} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

where the dimensions of $\mathbf{1}$ are implied from the context. We refrain from using $\boldsymbol{I}$ to represent the identity matrix in this paper to avoid confusion with the moment of inertia tensor matrix traditionally assigned the same letter. We denote the determinant of a square matrix $\boldsymbol{A}$ by $|\boldsymbol{A}|$ where $|\boldsymbol{A}| = 0$ implies singularity. We further denote the multiplicative inverse of non-singular square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ by $\boldsymbol{A}^{-1}$, such that $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \mathbf{1}$.

We denote the pre-multiplication of a row vector $\boldsymbol{x} \in \mathbb{R}^n$ by matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ to yield vector $\boldsymbol{y} \in \mathbb{R}^n$ by $\boldsymbol{A}\boldsymbol{x}^T = \boldsymbol{y}^T$. We often simplify the notation to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$, assuming an implicit transposition to befit matrix multiplication. We also adopt the vector transpose operator for matrices, denoting the transpose of a matrix $\boldsymbol{A}$ by $\boldsymbol{A}^T$ such that, element-wise $\left[a_{ij}\right]^T = \left[a_{ji}\right]$.

We denote sets as type subsets or subsets of other sets, optionally specifying qualifying conditions. We simply state a set $\boldsymbol{U}$ as a subset of set $\boldsymbol{V}$ using the notation $\boldsymbol{U} \subseteq \boldsymbol{V}$. We distinguish $\boldsymbol{U}$ as a *proper* subset of $\boldsymbol{V}$ using $\boldsymbol{U} \subsetneq \boldsymbol{V}$. We denote set equality using $\boldsymbol{U} = \boldsymbol{V}$ such that $\boldsymbol{U} \subseteq \boldsymbol{V}$ and $\boldsymbol{V} \subseteq \boldsymbol{U}$. We specify a member element u of $\boldsymbol{U}$ using $u \in \boldsymbol{U}$. We define a set $\boldsymbol{U}$, subset of $\boldsymbol{V}$ based on qualifying conditions using the notation $\boldsymbol{U} = \{v \in \boldsymbol{V} | p(v)\}$ where $p(v)$ is a predicate $p\colon \boldsymbol{V} \longmapsto \mathbb{B}$ that qualifies each element $v \in \boldsymbol{V}$ for membership or otherwise in $\boldsymbol{U}$. For example the set $\boldsymbol{U} = \{\boldsymbol{x} \in \mathbb{R}^2 \big| \boldsymbol{x}^{\mathbf{2}} \leq 9\}$ represents a circular region of radius three centred on the origin in $\mathbb{R}^2$. We hence denote the intersection and union of sets $\boldsymbol{U}$ and $\boldsymbol{V}$ by $\boldsymbol{U} \cap \boldsymbol{V}$ and $\boldsymbol{U} \cup \boldsymbol{V}$ such that $\boldsymbol{U} \cap \boldsymbol{V} = \{x | (x \in \boldsymbol{U}) \wedge (x \in \boldsymbol{V})\}$ and $\boldsymbol{U} \cup \boldsymbol{V} = \{x | (x \in \boldsymbol{U}) \vee (x \in \boldsymbol{V})\}$ where $\wedge$ and $\vee$ represent a Boolean conjunction and disjunction respectively. We similarly denote the difference of sets $\boldsymbol{U}$ and $\boldsymbol{V}$ by $\boldsymbol{U} \backslash \boldsymbol{V} = \{x | (x \in \boldsymbol{U}) \wedge (x \notin \boldsymbol{V})\}$. The empty set, whose type is implied by the context, is denoted by the symbol $\emptyset$, such that for any set $\boldsymbol{U}$, $\boldsymbol{U} \cup \emptyset = \boldsymbol{U}$ and $\boldsymbol{U} \cap \emptyset = \emptyset$. The cardinality, or size, of a set $\boldsymbol{U}$ is denoted by $|\boldsymbol{U}|$. Finally, we denote the set of all subsets of a set $\boldsymbol{U}$, termed the *power set* of $\boldsymbol{U}$, by $\mathcal{P}(\boldsymbol{U})$.

We define functions in terms of their type or set mapping, such as $f\colon \mathbb{R}^3 \longmapsto \mathbb{R}$, and in terms of their element-wise definition, such as $f(\boldsymbol{x}) = |\boldsymbol{x}|$. We denote the inverse of an invertible function f where $f\colon \boldsymbol{A} \longmapsto \boldsymbol{B}$, by f^{-1} such that $f^{-1}\colon \boldsymbol{B} \longmapsto \boldsymbol{A}$ and $f(a) = b \Leftrightarrow f^{-1}(b) = a$ for all $a \in \boldsymbol{A}$ and $b \in \boldsymbol{B}$. We denote the composition of two functions $f\colon \boldsymbol{A} \longmapsto \boldsymbol{B}$ and $g\colon \boldsymbol{B} \longmapsto \boldsymbol{C}$ by $g \circ f$, such that $g \circ f(a) = g\big(f(a)\big) = g(b) = c$ where $a \in \boldsymbol{A},$ $b \in \boldsymbol{B}$ and $c \in \boldsymbol{C}$.

For the conventions adopted on quaternion notation we refer the reader to Appendix A.

Appendix A
Quaternions

Quaternions were first described in 1843 by Irish mathematician Sir William R. Hamilton in his search for a higher-dimensional equivalent to complex numbers [**78**]. In the realm of graphics and physical simulation, quaternions find application as rotational transformations that offer a number of advantages over more traditional mathematical constructs such as matrices. Within this section, we provide a concise description of quaternions, together with the associated algebra and applications.

Quaternions are essentially 4D vectors whose space is defined over the field of real numbers $\mathbb{R}$, denoted by $\mathbb{R}^4 \stackrel{\text{def}}{=} \mathbb{H}$. A quaternion $\boldsymbol{q} \in \mathbb{H}$ may be written in the form $\boldsymbol{q} = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ where $w \in \mathbb{R}$ is the real component, and $x, y, z \in \mathbb{R}$ are the *imaginary* components of $\boldsymbol{q}$. A more compact representation is $\boldsymbol{q} = (s, \boldsymbol{v})$ where $s \in \mathbb{R}$ and $\boldsymbol{v} \in \mathbb{R}^3$ are, respectively, the real and imaginary components of $\boldsymbol{q}$ analogous to w and $[x, y, z]$. The compact representation will be subsequently used for convenience.

The addition of two quaternions $\boldsymbol{q}_1, \boldsymbol{q}_2 \in \mathbb{H}$ is defined by the component-wise addition of their elements

$$\boldsymbol{q}_1 + \boldsymbol{q}_2 = (s_1, \boldsymbol{v}_1) + (s_2, \boldsymbol{v}_2) = (s_1 + s_2, \boldsymbol{v}_1 + \boldsymbol{v}_2) \tag{A.1}$$

The quaternion $\boldsymbol{0} \in \mathbb{H}$, defined by $\boldsymbol{0} = (0, \boldsymbol{0})$, is termed the *additive identity* and is such that $\boldsymbol{q} + \boldsymbol{0} = \boldsymbol{0} + \boldsymbol{q} = \boldsymbol{q}$ for any $\boldsymbol{q} \in \mathbb{H}$.

The *additive inverse* of a quaternion $\boldsymbol{q} = (s, \boldsymbol{v})$, denoted by $-\boldsymbol{q}$ is defined by its component-wise additive inverses

$$-\boldsymbol{q} = -(s, \boldsymbol{v}) = (-s, -\boldsymbol{v}) \tag{A.2}$$

Hence, subtraction of quaternion $\boldsymbol{q}_2$ from $\boldsymbol{q}_1$ is defined by

$$\boldsymbol{q}_1 - \boldsymbol{q}_2 = \boldsymbol{q}_1 + (-\boldsymbol{q}_2) \tag{A.3}$$

from which follows that $\boldsymbol{q} - \boldsymbol{q} = \boldsymbol{0}$ for any $\boldsymbol{q} \in \mathbb{H}$.

The quaternion components $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ obey multiplication rules analogous to the rules of the imaginary components of complex numbers and hence

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -1 \tag{A.4a}$$

In addition, $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ also behave like a set of orthonormal 3D vectors under the vector product, thus

$$\boldsymbol{ij} = \boldsymbol{k}, \quad \boldsymbol{jk} = \boldsymbol{i}, \quad \boldsymbol{ki} = \boldsymbol{j}, \quad \boldsymbol{ji} = -\boldsymbol{k}, \quad \boldsymbol{kj} = -\boldsymbol{i}, \quad \boldsymbol{ik} = -\boldsymbol{j} \tag{A.4b}$$

The rules defined by (A.4a) and (A.4b) motivate the definition of multiplication of quaternion $\boldsymbol{q}_1 = w_1 + x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k}$ with $\boldsymbol{q}_2 = w_2 + x_2\boldsymbol{i} + y_2\boldsymbol{j} + z_2\boldsymbol{k}$ using the distributive law for multiplication and collecting terms to yield

$$\begin{aligned}\boldsymbol{q}_1\boldsymbol{q}_2 \quad &= (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) \\ &+(w_1x_2 + w_2x_1 + y_1z_2 - z_1y_2)\boldsymbol{i} \\ &+(w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)\boldsymbol{j} \\ &+(w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2)\boldsymbol{k}\end{aligned} \tag{A.5a}$$

Using the compact notation $\boldsymbol{q}_1 = (s_1, \boldsymbol{v}_1)$ and $\boldsymbol{q}_2 = (s_2, \boldsymbol{v}_2)$, quaternion multiplication may be defined more succinctly by

$$\boldsymbol{q}_1\boldsymbol{q}_2 = (s_1, \boldsymbol{v}_1)(s_2, \boldsymbol{v}_2) = (s_1s_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2, s_1\boldsymbol{v}_2 + s_2\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2) \tag{A.5b}$$

where $\boldsymbol{v}_1 \cdot \boldsymbol{v}_2$ and $\boldsymbol{v}_1 \times \boldsymbol{v}_2$ denote, respectively, the *scalar* and *vector product* of $\boldsymbol{v}_1$ with $\boldsymbol{v}_2$. In the special case where $\boldsymbol{v}_1 = \boldsymbol{0}$, quaternion multiplication degenerates to pre-multiplication of $\boldsymbol{q}_2$ by scalar s_1. By similar argument, $\boldsymbol{v}_2 = \boldsymbol{0}$ yields a post-multiplication of $\boldsymbol{q}_1$ by scalar s_2. In general, for scalar $a \in \mathbb{R}$ and quaternion $\boldsymbol{q} = (s, \boldsymbol{v})$, $a\boldsymbol{q} = (as, a\boldsymbol{v}) = (sa, \boldsymbol{v}a) = \boldsymbol{q}a$. Hence, it follows that $1 = (1, \boldsymbol{0})$ is the *multiplicative identity* of quaternions.

The equivalent of vector magnitude for a quaternion $\boldsymbol{q}$, termed the quaternion *norm*, is denoted by $|\boldsymbol{q}|$ and is defined as

$$|\boldsymbol{q}| = |s, \boldsymbol{v}| = \sqrt{s^2 + \boldsymbol{v}^2} \tag{A.6}$$

where $\boldsymbol{v}^2 = \boldsymbol{v} \cdot \boldsymbol{v}$ is scalar product of $\boldsymbol{v}$ with itself. In addition, the norm exhibits the property

$$|\boldsymbol{q}_1\boldsymbol{q}_2| = |\boldsymbol{q}_1||\boldsymbol{q}_2| \tag{A.7}$$

Like complex numbers, quaternion algebra also defines the *conjugate* of a quaternion $\boldsymbol{q}$, denoted by $\boldsymbol{q}^*$, as

$$\boldsymbol{q}^* = (s, \boldsymbol{v})^* = (s, -\boldsymbol{v}) \tag{A.8}$$

From (A.8) it follows that

$$\boldsymbol{q}\boldsymbol{q}^* = (s, \boldsymbol{v})(s, -\boldsymbol{v}) = (s^2 + \boldsymbol{v}^2, \boldsymbol{0}) = |\boldsymbol{q}|^{\boldsymbol{2}} \in \mathbb{R} \tag{A.9}$$

An important result concerning conjugates is that

$$(\boldsymbol{q}_1\boldsymbol{q}_2)^* = \boldsymbol{q}_2{}^*\boldsymbol{q}_1{}^* \tag{A.10}$$

The *multiplicative inverse* of quaternion $\boldsymbol{q} \neq \boldsymbol{0}$, denoted by $\boldsymbol{q}^{-1}$, is a quaternion such that $\boldsymbol{q}\boldsymbol{q}^{-1} = \boldsymbol{q}^{-1}\boldsymbol{q} = 1$ for all $\boldsymbol{q} \in \mathbb{H}$. Equations (A.6) and (A.9) provide a means for computing $\boldsymbol{q}^{-1}$ in terms of $\boldsymbol{q}$ as follows

$$\boldsymbol{q}^{-1} = \boldsymbol{q}^{*} \frac{1}{|\boldsymbol{q}|^{2}} \tag{A.11}$$

For a quaternion $\widehat{\boldsymbol{q}}$ such that $|\widehat{\boldsymbol{q}}| = 1$, termed a *unit quaternion*, it follows that $\widehat{\boldsymbol{q}}^{-1} = \widehat{\boldsymbol{q}}^{*}$ and that $\widehat{\boldsymbol{q}}\widehat{\boldsymbol{q}}^{*} = 1$.

The derivative of parameterised quaternion $\boldsymbol{q}(t) = \big(s(t), \boldsymbol{v}(t)\big) \in \mathbb{H}$ with respect to $t \in \mathbb{R}$ is defined in terms of the component-wise derivatives

$$\frac{d\boldsymbol{q}}{dt} = \left(\frac{ds}{dt}, \frac{d\boldsymbol{v}}{dt}\right) \tag{A.12}$$

Similarly, the indefinite integral of $\boldsymbol{q}(t)$ is defined as

$$\int \boldsymbol{q}\, dt = \left(\int s\, dt\, , \int \boldsymbol{v}\, dt\right) \tag{A.13}$$

Appendix B
Sample Gravitas Sandbox Simulation Script

The following is a sample script for setting up a simple configuration involving two unit spheres connected by an angular motor and allowed to move on a ground plane.

```
Geometries
{
  Geometry { Type = "Halfspace" Id = "GroundHalfspace" CustomProperties { } }

  Geometry { Type = "Sphere" Id = "UnitSphere" CustomProperties { Radius = 0.5 } }

  Geometry
  {
     Type = "Cylinder" Id = "MyCylinder"
     CustomProperties { Radius = 0.6 Height = 1.2 }
  }
}

Materials
{
  Material { Id  = "Ground" RestitutionCoefficient = 0.1
     SurfacePerturbationCoefficient = 0.1 }

  Material { Id  = "Plastic" RestitutionCoefficient = 0.4
     SurfacePerturbationCoefficient = 0.1 }
 }

Forces
{
  Force
  {
     Type = "Gravity" Id = "NormalGravity"
     CustomProperties { Acceleration = {0.0, -5.0, 0.0} }
  }
}

Simulator { Type= "Basic" Id = "DefaultSimulator" CustomProperties { } }

Environment
{
  Id = "DefaultEnvironment"

  Space
  {
     Type  = "Grid"
     Id    = "DefaultSpace"

     CustomProperties
     {
     }

     Bodies
     {
        Body
        {
           Type     = "Rigid" Id = "Ground"
           GeometryId = "GroundHalfspace" BoundingVolumeType = "None"
           MassDensity = 0.0 Material = "Ground"

           KineticProperties
           {
              Position = {0.0, 0.0, 0.0} Orientation = {1.0, 0.0, 0.0, 0.0}
              LinearVelocity = {0.0, 0.0, 0.0} AngularVelocity = {0.0, 0.0, 0.0}
              DampingCoefficient = 0.1 CanSleep = true
           }

           Forces = { }
        }

        Body
        {
           Type     = "Rigid" Id = " Ball01"
           GeometryId = " UnitSphere " BoundingVolumeType = "Sphere"
           MassDensity = 0.0 Material = "Plastic"
```

```
            KineticProperties
            {
               Position = {0.0, 1.0, 0.0} Orientation = {1.0, 0.0, 0.0, 0.0}
               LinearVelocity = {0.0, 0.0, 0.0} AngularVelocity = {0.0, 0.0, 0.0}
               DampingCoefficient = 0.1 CanSleep = true
            }

            Forces = { "NormalGravity" }
         }

         Body
         {
            Type     = "Rigid" Id = " Ball02"
            GeometryId = " UnitSphere " BoundingVolumeType = "Sphere"
            MassDensity = 0.0 Material = "Plastic"

            KineticProperties
            {
               Position = {0.0, 3.0, 0.0} Orientation = {1.0, 0.0, 0.0, 0.0}
               LinearVelocity = {0.0, 0.0, 0.0} AngularVelocity = {0.0, 0.0, 0.0}
               DampingCoefficient = 0.1 CanSleep = true
            }

            Forces = { "NormalGravity" }
         }
      }
   }

   Constraints
   {
      Constraint
      {
         Type  = "AngularMotor"
         Id    = "MyAngularMotor"

         Body1 = "Ball01"
         Body2 = "Ball02"

         BreakingThreshold = 100.0

         CustomProperties
         {
            BindAxisPoint1 = { 0.0, 1.0, 0.0 }
            BindAxisPoint2 = { 0.0, 3.0, 0.0 }
            AngularSpeed   = 3.1412
         }
      }
   }

   KineticIntegrator = "Euler"

   ConstraintSolver  = "Sequential"
}
```

Bibliography

[1] G. Booch, R. Martin, and J. Newkirk, *Object-oriented analysis and design with applications*, 3rd ed. Harlow, United Kingdom: Addison-Wesley, 1999.

[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design patterns : elements of reusable object-oriented software*. Reading, Mass, United Kingdom: Addison-Wesley, 1994.

[3] K. Buġeja, "Vistas : An extensible, object-oriented, real-time three-dimensional rendering framework," Master in Information Technology Thesis, University of Malta, Msida, 2008.

[4] B. Schaeffer. (2006) Armadillo Run. [Online]. http://www.armadillorun.com/

[5] (2005) Chronic Logic. [Online]. www.chroniclogic.com/gish.htm

[6] Switchball. [Online]. http://www.switchball.com/

[7] D. H. Eberly, "Lagriangian Dynamics," in *Game Physics*. San Francisco, CA94111: Morgan Kaufmann, 2004.

[8] G. K. Mangion, "Cuneus : An Embeddable Scripting Framework based on a virtual machine paradigm," Master in Information Technology Thesis, University of Malta, Msida, 2008.

[9] J. Buck, "Rally Cross Physics," International Game Developers Association Presentation, 2007.

[10] D. H. Eberly, "Mass-Spring Systems," in *Game Physics*. San Francisco, CA 94111: Morgan Kaufmann, 2004, ch. 4, pp. 164-172.

[11] K. Erleben, J. Sporring, K. Henriksen, and H. Dolmann, "The Finite Element Method," in *Physics-Based Animation*. Hingham, Massachusetts 02043: Charles River Media, ch. 10, pp. 335-370.

[12] C. Ericson, *Real-Time Collision Detection*, T. Cox, Ed. San Francisco, CA 94111: Morgan Kaufmann, 2005, pp. 64,101-103,106,156-160.

[13] D. Baraff. (1997) School of Computer Science, Carnegie Mellon University. [Online]. http://www.cs.cmu.edu/~baraff/sigcourse/notesd2.pdf

[14] S. Redon, A. Kheddar, and S. Coquillart, "Fast continuous collision detection between rigid bodies," *Computer Graphics Forum*, vol. 21, no. 3, pp. 279-287, May 2003.

[15] X. Zhang, S. Redon, M. Lee, and Y. J. Kim, "Continuous collision detection for articulated models using Taylor models and temporal culling," ACM SIGGRAPH 2007 paper, SIGGRAPH, San Diego, California , 2007.

[16] B. V. Mirtich, "Impulse-based Dynamic Simulation of Rigid Body Systems," Phd Thesis, University of California, Berkeley, 1996.

[17] C. Ericson, "Spatial Partitioning," in *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2005, ch. 7, pp. 285-348.

[18] C. Ericson, "BSP Tree Hierarchies," in *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2005, ch. 8, pp. 349-382.

[19] C. Ericson, "Bounding Volume Hierarchies," in *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann, 2005, ch. 6, pp. 235-284.

[20] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking,"

vol. 22, no. 3, pp. 871-878, Jul. 2003.

[21] A. A. Schmitt, "A Dynamical Simulation of Linked Rigid Body Systems using Impulse Technique," University Karlsruhe, 2003.

[22] R. Smith. (2008) Open Dynamics Engine. [Online]. http://www.ode.org/

[23] P. Lab. (2008) Simple Physics Engine. [Online]. http://spehome.com/

[24] E. Coumans. (2008) Bullet Physics. [Online]. http://www.bulletphysics.com

[25] A. H. Watt, *3D Computer Graphics*, 3rd ed. Reading, Harlow, England: Addison Wesley, 1999.

[26] A. F. Möbius, *Der baryzentrische Calcul* . Leipzig, 1827.

[27] C. F. V. L. Gene H Golub, *Matrix Computations*, 3rd ed. Baltimore, USA: Johns Hopkins University Press, 1996.

[28] Ken Shoemake, The Singer Company, Link Flight Simulation Division, 1700 Santa Cruz Ave., Menlo Park, CA, "Animating rotation with quaternion curves," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1985, pp. 245-254.

[29] J. L. R. D'Alembert, *Traité de dynamique*. Paris, 1743.

[30] D. H. Eberly. (2008, Mar.) Geometric Tools. [Online]. http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

[31] B. V. Mirtich, "Fast and accurate computation of polyhedral mass properties," *Journal of Graphics Tools*, vol. 1, no. 2, pp. 31-50, Feb. 1996.

[32] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, PA, USA: SIAM: Society for Industrial and Applied Mathematics, 1998.

[33] G. Dahlquist, "A Special Stability Problem for Linear Multistep Methods," *BIT*, vol. 3, no. 1, pp. 27-43, 1963.

[34] L. Verlet, "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," *Physical Review*, vol. 159, no. 1, pp. 98-103, Jan. 1967.

[35] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, 2nd ed. Wiley, 2003.

[36] C. Ericson, "Testing Sphere Against AABB," in *Real-Time Collision Detection*, T. Cox, Ed. San Francisco, CA 94111: Morgan Kaufmann, 2005, ch. 5, pp. 165-166.

[37] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193-203, Apr. 1988.

[38] R. J. Gardner, "The Brunn-Minkowski inequality," *Bulletin of the American Mathematical Society*, vol. 39, no. 3, pp. 358-359, Apr. 2002.

[39] J. Canny, "Collision Detection for Moving Polyhedra," Massachusetts Institute of Technology, Cambridge, Technical Report AIM-806 , 1984.

[40] E. Schomer and C. Thiel, "Efficient collision detection for moving polyhedra," in *11th Annul ACM Symposium on Computational Geometry*, Vancouver, British Columbia, Canada , 1995, pp. 51-60.

[41] B. Kim and J. Rossignac, "Collision prediction for polyhedra under screw motions," in *Proceedings of the eighth ACM symposium on Solid modeling and applications*, Seattle,

Washington, USA, 2003, pp. 4-10.

[42] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann, "Multiple Points of Collision," in *Physics-Based Animation*. Hingham, Massachusetts 02043: Charles River Media, Inc., 2005, ch. 6, p. 152.

[43] G. van den Bergen, "Distance and Penetration Depth Computation," in *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003, ch. 4, pp. 115-120.

[44] G. van den Bergen, "Penetration Depth," in *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003, ch. 4, pp. 147-170.

[45] G. van den Bergen. (2001) Technische Universiteit Eindhoven: Wiskunde & Informatica. [Online]. http://www.win.tue.nl/~gino/solid/gdc2001depth.pdf

[46] C.-A. d. Coulomb, *Théorie des machines simples, en ayant égard au frottement de leurs parties et à la roideur des cordages*. 1779.

[47] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann, "The Permissible Region in Impulse Space," in *Physics-Based Animation*. Hingham, Massachusetts 02043: Charles River Media, 2005, ch. 6, pp. 139-149.

[48] X. Provot, "Deformation constraints in a mass-spring model to describe rigid cloth behavior," Institut National de Recherche en Informatique et Automatique France, 1995.

[49] S. Hasegawa, N. Fujii, Y. Koike, and M. Sato, "Real-Time Rigid Body Simulation Based on Volumetric Penalty Method," *Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2003*, p. 326, Mar. 2003.

[50] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann, "Solving Harmonic Oscillators Numerically," in *Physics-Based Animation*. Hingham, Massachusetts 02043: Charles River Media, 2005, ch. 5, pp. 104-106.

[51] R. Barzel and A. H. Barr, "A modeling system based on dynamic constraints," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, Pasadena, CA, 1988, pp. 179-188.

[52] C. Pedersen, K. Erleben, and J. Sporring, "Ballet balance strategies," *Simulation Modelling Practice and Theory*, vol. 14, no. 8, pp. 1135-1142, Nov. 2006.

[53] M. Moore and J. Wilhelms, "Collision detection and response for computer animation," *Computer Graphics (Proceedings of SIGGRAPH 88)*, vol. 22, pp. 289-298, 1988.

[54] D. Baraff, "Fast Contact Force Computation for," in *Computer Graphics Proceedings, Annual Conference Series, 1994*, Orlando, 1994, pp. 23-34.

[55] J. Cottle, S. Pang, and R. E. Stone, *The Linear Complementarily Problem*. Academic Press Inc., 1992.

[56] N. Megiddo, "A Note on the Complexity of P-Matrix LCP and Computimg an Equilibrium," IBM Almaden Research, San Jose, CA 95120-6099, Research Report RJ6439, 1988.

[57] R. Featherstone and D. Orin, "Robot Dynamics: Equations and Algorithms," in *Prodceedings, IEEE International Conference of Robotics & Automation*, San Francisco, CA, 2000, p. 826{834.

[58] E. Kokkevis, "Practical Physics for Articulated Characters," Research & Development, Sony Computer Entertainment America Game Developers Conference Presentation, 2004.

[59] Havok. (2008) Havok Physics. [Online]. http://www.havok.com/content/view/17/30/

[60] Newton Game Dynamics. (2007) Newton Game Dynamics. [Online]. http://www.newtondynamics.com/

[61] Nvidia Corporation. (2008) PhysX and AGEIA Physx Accelerator. [Online]. http://www.nvidia.com/object/nvidia_physx.html

[62] D. K. Arrowsmith and C. M. Place, "Section 3.3," in *Dynamical Systems*. London, United Kingdom: Chapman & Hall, 1992, ch. 3.

[63] G. George and B. K. Guenter, "An Object-Oriented Architecture for the Simulation of Rigid-Body Kinematics," Georgia Tech College of Computing Experience Paper, 1993.

[64] E. Catto, "Modeling and Solving Constraints," Game Developers Conference 2008 Presentation, 2008.

[65] J. W. Baumgarte, "A new method of stabilization for holonomic constraints," *Transactions, Journal of Applied Mechanics (ISSN 0021-8936)*, vol. 50, no. 4a, p. 869,870, Dec. 1983.

[66] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Indianapolis, IN 46290: Addison-Wesley, 2000.

[67] B. Hayes, "A Lucid Interval," *American Scientist*, vol. 91, no. 6, p. 487, Dec. 2003.

[68] R. Barzel, J. F. Hughes, and D. N. Wood, "Plausible Motion Simulation for Computer Graphics Animation," in *Eurographics Workshop Proceedings, Computer Animation and Simulation '96*, New York, 1996, pp. 183-197.

[69] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, "Observer," in *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass, United Kingdom: Addison-Wesley, 1994, ch. 5, p. 293.

[70] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, "Singleton," in *Design patterns : elements of reusable object-oriented software*. Reading, Mass, United Kingdom: Addison-Wesley, 1994, ch. 3, p. 127.

[71] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, "Strategy," in *Design patterns : elements of reusable object-oriented software*. Reading, Mass, United Kingdom: Addison-Wesley, 1994, ch. 5, p. 315.

[72] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, "Abstract Factory," in *Design patterns : elements of reusable object-oriented software*. Reading, Mass, United Kingdom: Addison-Wesley, 1994, ch. 3, p. 87.

[73] Microsoft Corporation. (2008) DirectX Developer Centre. [Online]. http://msdn.microsoft.com/directx

[74] Khronos Group. (2008) OpenGL. [Online]. http://www.opengl.org/

[75] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 1983.

[76] P. Bourke. (2008) University of Western Austrialia. [Online]. http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/

[77] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi, "I-COLLIDE: An interactive and exact collision detection system for large-scale environments," in *Symposium on Interactive 3D Graphics*, 1995, pp. 189-196.

[78] W. R. Hamilton, *Lectures on quaternions*. Ithaca, New York: Cornell University Library, 1853.