REAL-TIME DYNAMICS FOR INTERACTIVE ENVIRONMENTS

A Thesis

by

ALEXANDER NIKOLAI TIMCHENKO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2007

Major Subject: Visualization Sciences

REAL-TIME DYNAMICS FOR INTERACTIVE ENVIRONMENTS

A Thesis

by

ALEXANDER NIKOLAI TIMCHENKO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee, Donald House
Committee Members, Frederic Parke
John Keyser

Head of Department, Mark Clayton

December 2007

Major Subject: Visualization Sciences

ABSTRACT

Real-time Dynamics for Interactive Environments. (December 2007)

Alexander Nikolai Timchenko, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Donald House

This thesis examines the design and implementation of an extensible object-oriented physics engine framework. The design and implementation consolidates concepts from the wide literature in the field and clearly documents the procedures and methods. Two primary dynamic behaviors are explored: rigid body dynamics and articulated dynamics. A generalized collision response model is built for rigid bodies and articulated structures which can be adapted to other types of behaviors.

The framework is designed around the use of interfaces for modularity and easy extensibility. It supports both a standalone physics engine and a supplement to a distributed immersive rendering environment. We present our results as a number of scenarios that demonstrate the viability of the framework. These scenarios include rigid bodies and articulated structures in free-fall, collision with dynamic and static bodies, resting contact, and friction. We show that we can effectively combine different dynamics into one cohesive structure. We also explain how we can efficiently extend current behaviors to develop new ones, such as altering rigid bodies to produce different collision responses or flocking behavior. Additionally, we demonstrate these scenarios in both the standalone and the immersive environment.

To my parents

ACKNOWLEDGMENTS

I would like to thank the members of my committee: my chair, Dr. Donald House, for his help and guidance with this thesis; Dr. Frederic Parke for the Immersive Visualization Project at Texas A&M University and his help with the immersive engine; and Dr. John Keyser for introducing me to physically based simulation. I would also like to thank all of the faculty, staff, and students of the Visualization Program for their advice and assistance. And finally, I would like to thank my parents to whom I dedicate this thesis. Without their love, guidance, and support I would not be where I am today.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

II PREVIOUS WORK 3

III BACKGROUND 8

A. Rigid Body Dynamics 8
1. Rigid Body Properties 8
2. Rigid Body State 12
3. Collision Response 13
B. Articulated Dynamics 17
1. Articulated Structures 18
2. Joint and Link Properties 20
3. Spatial Algebra 23
4. The Featherstone Articulated Body Method 27
5. Articulated Dynamics State 32
6. Collision Response 32
C. Immersive Engine 34

IV METHODOLOGY AND IMPLEMENTATION 37

A. Interfaces 38
1. `SimulationState` 38
2. `Integratable` 41
3. `Collidable` 43
4. `Constraint` 46
5. `Renderable` and `SidRenderable` 47
6. `NetSync` 49
B. The Physics Engine 51
1. The `Object` Class 51
2. The Simulation System 52
3. The Collision Detector 54
a. Collision Detection 56
b. Colliding Contact Resolution 59
c. Resting Contact Resolution 61

CHAPTER Page

d. Friction 62
4. The Simulation Step 66
C. GUPPY3D and Dynamics 67

V RESULTS AND DISCUSSION 70

A. Documentation of Concepts and Algorithms 70
B. The Framework 71
1. Rigid Bodies 71
2. Articulated Dynamics 72
3. Collision Detection and Response 75
4. Integration 77
5. Framework Extensibility 78
C. Dynamics in the Cave 81

VI CONCLUSION AND FUTURE WORK 86

REFERENCES 88

APPENDIX A 91

VITA 93

LIST OF FIGURES

FIGURE Page

1 Bounding volumes. 6

2 A force acting on a rigid body. 12

3 Rigid body contact. 14

4 An example of multiple resting contacts. 17

5 An articulated chain with a fixed base. 18

6 Revolute and prismatic joints. 19

7 Common vectors defined between links and joints. 20

8 A tree-like articulated structure. 29

9 Articulated body method. 30

10 Example cave geometries. 35

11 GUPPY3D and the physics engine. 37

12 Object hierarchy and interface diagram. 39

13 An example of `NetSync` structure. 50

14 Two sphere-trees in contact. 56

15 Sphere contact and intersection. 57

16 Third sphere-tree level of a dragon model. 58

17 Spinning sphere problem. 59

18 Contact with friction. 62

19 Rigid bodies in the standalone application. 72

FIGURE Page

20 Rigid bodies in resting contact. 73

21 Rigid bodies in a billiards simulation. 74

22 Bowling. 74

23 Various articulated scenarios. 75

24 Parent spheres do not cover all children spheres. 76

25 Block on an inclined plane. 77

26 Flocking behavior from the `Boid` class. 81

27 Flocks in GUPPY3D. 82

28 Billiards in GUPPY3D. 83

29 Bowling in GUPPY3D. 83

30 Rigid bodies in GUPPY3D. 84

31 Articulated dynamics in GUPPY3D. 84

CHAPTER I

INTRODUCTION

Simulating physical behavior is an important subfield of Computer Graphics and finds important uses in such areas as education, design, film and animation, and video games. In education and design, simulations can be used to study how virtual prototypes would behave in the real world. In film, effects like falling debris and explosions provide exciting visuals or bring amazing virtual creatures to life through articulated animation. In video games, physical behavior adds a whole new dimension of interaction as the player can utilize the environment to his advantage through realistic and plausible means.

There is a great body of literature on simulation topics ranging from particles and flocking to fluid dynamics. While the literature explores the concepts well, it often does not expose implementation details that are difficult and complex.

This thesis focuses on the design and implementation of a general purpose real-time dynamics framework. We address the implementation of two major aspects of dynamic interactions: rigid bodies and articulated dynamics. Many solid objects can be represented as rigid bodies. These rigid bodies can fall, bounce, roll, and tumble realistically. Articulated dynamics include simulating a wide array of behavior such as chains, machinery, and skeletal animation. The various dynamic behaviors can interact with each other and be affected by the user.

The framework is designed to function in isolation or as a component in an immersive rendering engine. A virtual immersive environment frequently consists of several screens and projectors that surround the user with an image of the virtual

---

The journal model is *IEEE Transactions on Automatic Control.*

world. The Immersive Visualization Project at Texas A&M is one such installation. However, the user interaction present in the proprietary display engine for this system is currently limited. The user can walk or fly through the world and affect it through scripted events. Addition of physically-based dynamic behaviors would serve to greatly increase the interactivity and believability of virtual environments.

During the design of our physics framework, we have run across a number of issues that are not clearly exposed in the literature. This thesis documents such issues and implementation details while presenting the basics necessary for understanding the fundamentals. We discuss rigid body and articulated dynamics concepts, including simulation and response to impulses and forces. Additionally, we illustrate how we designed the framework and dealt with topics such as simulating different types of dynamics in one system and collision response. We also address incorporating our system into an immersive rendering engine.

In summary, this thesis has the following goals:

1. Clearly document, from the point of view of the implementor, the procedures, algorithms, and concepts involved in rigid body and articulated dynamics.

2. Design and validate an extensible object-oriented physics engine framework that supports rigid body and articulated dynamics.

3. Wrap the framework in a standalone application as proof-of-concept that the framework is capable of supporting physics.

4. Supplement a distributed immersive rendering engine with the physics engine.

CHAPTER II

PREVIOUS WORK

Over the past decades, research in virtual reality and immersive visualization yielded new ways to view a virtual world. Two primary approaches have been developed: head-mounted devices (HMD's) and spatially immersive displays.

Early functional prototypes of VR helmets were available in the mid 1980s [1]. Such HMD's typically consisted of a stereoscopic display system that provided an image for the right and left eye using LCDs. The user's head position and orientation are used to determine the viewing position and angle of the virtual camera. Since the LCDs are essentially worn over the eyes, the user would see the world no matter how he turned his head. Interaction was further enhanced through control methods varying from traditional mouse and keyboard to haptics and speech and gesture recognition.

Spatially immersive displays offered a viable alternative to virtual reality HMD's. These systems typically consisted of several displays that surround a user or group of users. A spatially coherent world was rendered to these displays, minimizing seams as much as possible. One of the first prototypes of such an immersive system was designed at the Electronic Visualization Laboratory at the University of Illinois [2]. The system was named the CAVE and was made up of four rectangular facets in the shape of a cube making up three walls and a floor. Back-projection was used for the walls and top-down projection for the floor. Similar installations of four to six facets are popular today.

Over the last decade, SID's have undergone a great deal more research and advancement. Projects like the Immersive Visualization Project at Texas A&M [3] aim to significantly reduce the costs of producing an immersive system. The system consists of modular screens, commodity projectors, and readily available hardware. The

screens can be assembled to approximate spheres, cylinders, and other surrounding shapes. The proprietary engine driving the system allows the user to navigate worlds that can be modeled and textured using standard modeling packages such as Alias Maya or Blender.

This Immersive Visualization Project goes a long way to proving the viability of more affordable commodity SID's. However, users currently have limited interaction with the virtual world. They can fly or walk through the world and affect it through scripted means. Dynamic aspects can add a very powerful immersive element to such a system.

One common way of representing objects in a virtual world is through solid masses. Having such objects bounce, roll, and tumble realistically is an important factor in producing a believable environment. Physically-based rigid body simulation attempts to do just that.

Using well established concepts of classical mechanics, the motion of rigid objects can be simulated by tracking mass, inertial tensors, positions, rotations, and linear and angular momenta [4]. This allows solid objects to fall through the air realistically and react to the forces of wind and gravity. By determining the point(s) of contact and applying appropriate forces or impulses, rigid bodies can be made to collide, tumble, roll, and come to rest realistically.

Some dynamic structures are difficult to simulate using only rigid bodies. If modeled using impulse methods, hinged and prismatic joints generate constant collisions. Attempting to resolve all collisions can bring a simulation to a halt. Instead, such joints can be modeled using constraints to keep bodies properly attached.

Several approaches have been developed to deal with joint constraints. These approaches fall into two general categories: maximal coordinate and reduced coordinate methods [5]. Maximal coordinate methods keep track of each rigid body link in an

articulated structure separately. Lagrangian systems of constraints are then enforced to reduce the degrees of freedom. Linear run-time versions of these methods, such as the Recursive Newton-Euler Algorithm [6, 7], were developed for use in real-time applications.

Reduced coordinate methods, on the other hand, deal with the joint angles of an articulated structure directly. Everything is expressed in local coordinates; constraints are guaranteed because the unwanted degrees of freedom are removed from the simulation. The equations of motion can be derived manually by hand for simple problems or at run-time by more advanced methods such as Featherstone's Articulated Body Method [8, 9]. Additionally, such a system does not suffer from numeric drift as Lagrange methods tend to. No stabilization terms are therefore necessary.

Rigid bodies and articulated bodies often interact with each other as well as world and user geometry. Therefore, the two problems that need to be solved are when and where collisions occur (collision detection) and what to do with this information to prevent bodies from interpenetrating (collision response).

Collision detection between rigid objects is computationally complex, $O(n^2)$ in the number of collidable surfaces. Therefore, methods of speeding up this process have been developed [4]. Convex polyhedra can be fairly quickly tested for collisions using separating planes. If bodies are not undergoing fast rotation, a face of one object or a pair of edges will often separate the two objects. The existence of a separating plane means the bodies cannot be intersecting. If one cannot be found, the bodies must intersect, but not all implementations can find all separating planes.

Another common approach to determine if two bodies are colliding is use of hierarchies of bounding volumes. Common choices for bounding primitives include axis-aligned bounding boxes, object-oriented bounding boxes, and spheres. Figure 1 illustrates two alternatives for how these hierarchies can be constructed from bounding

boxes and spheres to form oriented bounding box trees and sphere-trees. Hierarchies have the benefit that if a parent volume does not intersect, none of its children will intersect. Therefore, parts of objects that are not intersecting can be quickly culled and ignored in further calculations. Publicly available libraries V-COLLIDE and RAPID are based on object-oriented bounding box hierarchies and offer significantly faster results [10, 11]. However, these software packages do not provide the actual collision point and normal data.

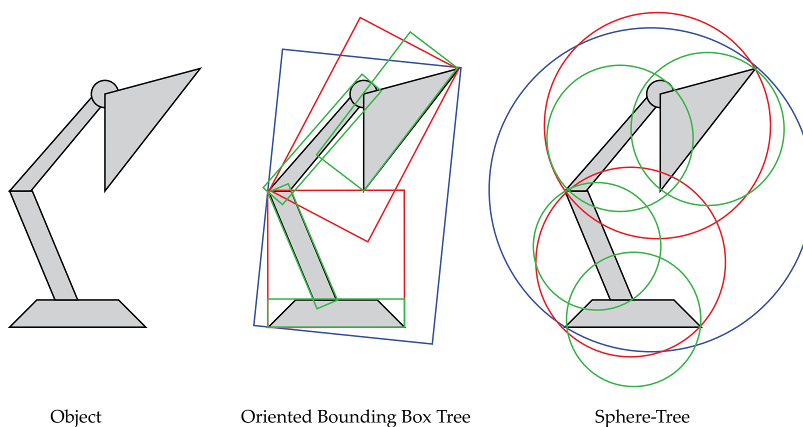

Fig. 1. Bounding volumes.
The object and its corresponding object-oriented bounding box tree and its sphere tree. The red volume is the first level, the blue is the second, and the green the third. Note that the children are not necessarily confined to the space of the parent.

Other methods of collision detection and determination using level sets have also been developed. A rigid body can be represented using a three-dimensional grid of distance values where positive values specify that a point is outside the body and negative values specify that a point is inside. This representation is essentially a signed distance function $\phi(x, y, z)$. The value itself gives an approximate distance to the surface and the gradient of the function, $\nabla\phi$, gives an approximate surface normal [12]. Fast marching methods such as those described by Sethian [13] can be used to

construct such a representation.

CHAPTER III

BACKGROUND

This chapter focuses on the concepts of rigid body simulation and articulated dynamics and briefly talks about the immersive engine. There is a strong body of literature established for rigid bodies and articulated dynamics and the thesis draws heavily on these concepts. We present a concise description of the problems and concepts, but for proofs and complex derivations, the reader is directed to original sources.

A. Rigid Body Dynamics

The concept of a rigid body representation is a very powerful one in dynamics. In reality, all objects deform when under the influence of contact impulses or forces. However, computing the tiny deformations is unnecessary for believable animation of rigid bodies.

To simulate motion of an unconstrained rigid body, we look to Newtonian dynamics. The study of rigid body motion is a well documented topic and this thesis utilizes Baraff and Witkin's SIGGRAPH 99 course notes on rigid body simulation for its implementation [4].

1. Rigid Body Properties

A rigid body in free-fall can be represented with several properties. At any one time, in order to display the rigid body, we need to know its position and orientation. We will store position as a three-dimensional translation vector $\mathbf{x}$ and the rotation as a unit quaternion $\mathbf{q}$. When a rotation is needed in matrix form, a 3×3 matrix $\mathbf{R}$ is generated from $\mathbf{q}$. Since a rigid body by definition cannot deform, we can apply these transformations to a fixed local coordinate system of the object called body

space, in which the rigid body is initially defined. Therefore, position and rotation become transformations from local coordinate space to world space. Additionally, we require that the center of mass of the rigid body is located at the origin of the body space. Since an unconstrained rigid object can only rotate about its center of mass, this requirement simplifies rotations. Using these properties, we can determine how a point in world space relates to a particular location on a rigid body and vice-versa.

After defining the position and orientation of the body, we need to know how these properties change over time. The first derivative of position is velocity $\mathbf{v}$ a three-dimensional vector describing the rate of change of the body's center of mass over time. This is a translational property, and therefore only affects the position, not the orientation. The rate of change of the orientation is given by the angular velocity. Angular velocity is expressed as $\boldsymbol{\omega}$ and is also stored as a three-dimensional vector. The direction of $\boldsymbol{\omega}$ is the axis of rotation following the right-hand rule. The magnitude of $\boldsymbol{\omega}$ is the speed, in radians per second, at which the body is rotating. The rate of change of linear velocity and angular velocity are linear acceleration $\mathbf{a}$ and angular acceleration $\boldsymbol{\alpha}$.

Next, we define mass properties. We will assume that the body is of constant density and has a scalar mass m. Additionally, we can temporarily assume that a rigid body is made up of a number of particles, each having mass m_i, such that

$$m = \sum_{i=1}^{N} m_i.$$

If each particle i is located at position $\mathbf{r}_i$ from the origin of the body space, the center of mass of the object is

$$\mathbf{x}_{com} = \frac{\sum_{i=1}^{N} m_i \mathbf{r}_i}{m}.$$

The origin of body space should be translated to $\mathbf{x}_{com}$ so that in body space, $\mathbf{x}_{com} = 0$.

We will need one more mass property for rigid bodies: the inertial tensor. The inertial tensor is a 3×3 matrix

$$\mathbf{I} = \sum_{i=1}^{N} \begin{bmatrix} m_i(\mathbf{r}_{iy}^2 + \mathbf{r}_{iz}^2) & -m_i(\mathbf{r}_{iy}\mathbf{r}_{ix}) & -m_i(\mathbf{r}_{iz}\mathbf{r}_{ix}) \\ -m_i(\mathbf{r}_{iy}\mathbf{r}_{ix}) & m_i(\mathbf{r}_{ix}^2 + \mathbf{r}_{iz}^2) & -m_i(\mathbf{r}_{iz}\mathbf{r}_{iy}) \\ -m_i(\mathbf{r}_{iz}\mathbf{r}_{ix}) & -m_i(\mathbf{r}_{iz}\mathbf{r}_{iy}) & m_i(\mathbf{r}_{ix}^2 + \mathbf{r}_{iy}^2) \end{bmatrix}$$

that determines how a body's angular momentum relates to its angular velocity.

For a true representation of the inertial tensor, we would need to convert the summation to an integral and evaluate it over the volume of the body. This, however, is very difficult to do for all but the simplest of shapes.

Additionally, the integral would have to be reevaluated every time the object rotated. Fortunately, we can compute the inertial tensor once initially in body space to get $\mathbf{I}_{body}$, which is constant. The inertial tensor, based on the object's current rotation, is given by

$$\mathbf{I} = \mathbf{R}\mathbf{I}_{body}\mathbf{R}^T.$$

Instead of storing velocities as part of the state, we use linear and angular momenta. Since a rotating body's angular velocity may change without external forces, using conservative momentum properties is more appropriate. The linear momentum of a rigid body is a three-dimensional vector

$$\mathbf{P} = m\mathbf{v},$$

giving the momentum of the body's center of mass. Newton's Second law then gives the rate of change of linear momentum,

$$\mathbf{F} = m\mathbf{a}\,,$$

$$\mathbf{F} = \dot{\mathbf{P}}\,,$$

which is simply the total force acting on the body. The angular momentum for a rigid body is the three-dimensional vector

$$\mathbf{L} = \mathbf{I}\boldsymbol{\omega}.$$

Analogous to the relationship between linear momentum and force, we have a relationship between angular momentum and torque $\boldsymbol{\tau}$,

$$\dot{\mathbf{L}} = \boldsymbol{\tau}.$$

In summary, a force will change the linear momentum of a body, affecting its linear velocity, and a torque will change the angular momentum of a body, affecting its angular velocity.

It is instructive to show how a rigid body reacts to an arbitrary force acting on an arbitrary point on the body. Say that a force $\mathbf{f}$ acts on a point $\mathbf{p}$ as shown in Figure 2. If the force is not applied in the direction of the center of mass or along the vector $\mathbf{r}$, the force will produce a torque,

$$\boldsymbol{\tau} = \mathbf{r} \times \mathbf{f},$$

where $\mathbf{r}$ is the vector from the center of mass to the point. Thus, the effects of the force on the linear and angular momenta are

$$\dot{\mathbf{P}} = \mathbf{f}$$

and

$$\dot{\mathbf{L}} = \mathbf{r} \times \mathbf{f}.$$

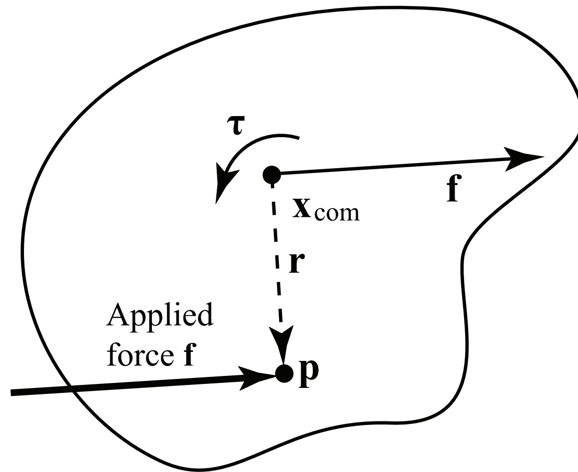

Fig. 2. A force acting on a rigid body.
Force $\mathbf{f}$ acting on $\mathbf{p}$ affects the center of mass via a force $\mathbf{f}$ and a torque $\boldsymbol{\tau}$.

2. Rigid Body State

The state of a rigid body is denoted as a vector of properties $\mathbf{X}$ and consists of position, rotation, linear momentum, and angular momentum. The state for a single rigid body is then defined as

$$\mathbf{X} = \begin{bmatrix} \mathbf{x} \\ \mathbf{q} \\ \mathbf{P} \\ \mathbf{L} \end{bmatrix}.$$

In order to simulate an object, we need to determine how state $\mathbf{X}$ is changing by calculating the derivative, $\dot{\mathbf{X}}$. The rate of change of a rotation quaternion is given by

$$\dot{\mathbf{q}} = \frac{1}{2}\boldsymbol{\omega}\mathbf{q}.$$

Thus, the rate of change of the system state in terms of the current state and applied

forces is

$$\dot{\mathbf{X}} = \begin{bmatrix} \mathbf{v} \\ \frac{1}{2}\boldsymbol{\omega}\mathbf{q} \\ \mathbf{F} \\ \boldsymbol{\tau} \end{bmatrix}.$$

In order to find the derivative of the current state of a rigid body, we need to know the linear and angular velocity given by

$$\mathbf{v} = \frac{1}{m}\mathbf{P}\,,$$
$$\boldsymbol{\omega} = \mathbf{I}^{-1}\mathbf{L}\,.$$

If we have an ordinary differential equation (ODE) solver, we can use the current state and the derivative of the state to simulate our rigid body.

3. Collision Response

Rigid body collision response is fairly straightforward. Given that two rigid bodies A and B are in contact, the collision detection routines should be able to provide a contact point $\mathbf{p}$ and a contact normal $\mathbf{n}$. Depending on how the simulation system is designed, the simulation might allow for interpenetrations between rigid bodies or the state of all bodies might have to be rolled back until a precise contact occurs.

Figure 3 depicts two bodies in exact contact, showing contact point $\mathbf{p}$ and the collision surface normal $\mathbf{n}$. The velocity at any point $\mathbf{p}$ on a rigid body is

$$\mathbf{v}_p = \mathbf{v} + \boldsymbol{\omega} \times \mathbf{r}_p,$$

where $\mathbf{r}_p$ is the vector from the origin of the body's coordinate system to $\mathbf{p}$ in body space. Therefore, relative speed along the normal of the bodies at the contact point

is

$$v_{rel} = \mathbf{v}_p^A \cdot \mathbf{n} - \mathbf{v}_p^B \cdot \mathbf{n}.$$

If v_{rel} is negative, then the bodies are moving towards each other at that point and will intersect on the next timestep if their velocities are not corrected. If v_{rel} is zero (to within some tolerance), then the bodies are in resting contact which has to be dealt with separately. If v_{rel} is greater than zero, then the bodies are separating.

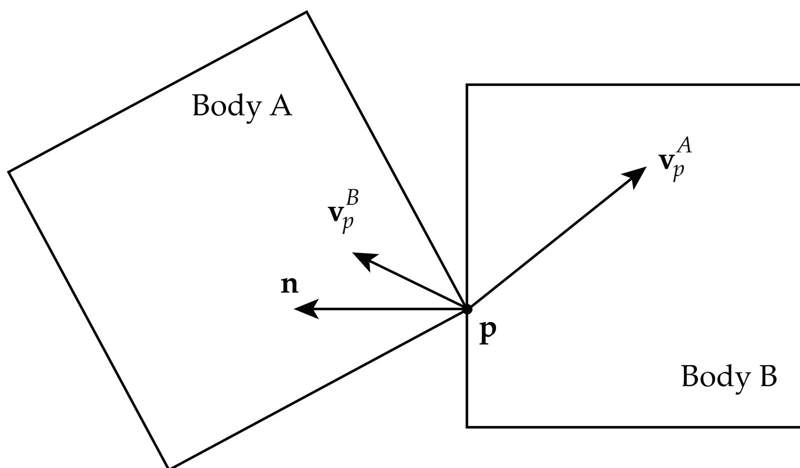

Fig. 3. Rigid body contact.
Body A is in contact with Body B at **p**. We know the normal **n** and the velocity of each of the bodies at the contact point.

To correct a negative relative velocity, we will apply an impulse to instantaneously change the relative velocity of the bodies involved. A coefficient of restitution ϵ is used to determine the resulting velocity of the collision:

$$v_{rel}^+ = -\epsilon v_{rel}^-.$$

Therefore, the change in velocity at the instant of collision is

$$\Delta v_{rel} = -(1 + \epsilon) v_{rel}^-.$$

We need to find an impulse of magnitude j that when applied to both bodies in opposite directions along the contact normal produces this change in velocity. For rigid bodies, the matrix

$$\mathbf{K} = \frac{1}{m} \cdot \mathbf{1} - \mathbf{r}\mathbf{I}^{-1}\mathbf{r},$$

where $\mathbf{1}$ is the 3×3 identity matrix, determines how a body responds to an impulse acting at a point a distance $\mathbf{r}$ from the center of mass. Thus, the magnitude of the impulse required to induce the desired velocity change is

$$j = \frac{1}{\mathbf{n}^T \mathbf{K} \mathbf{n}} \Delta v_{rel}.$$

During resting contact, the bodies are touching, but will not penetrate on the next time step. However, it is possible for the bodies to be accelerating towards each other and to gain an intersecting velocity on the next time step. Resting contact resolution attempts to solve this problem.

In the case of resting contact, we need to look at the relative acceleration a_{rel}^- of the point to determine if the contact is truly a resting contact. If a_{rel}^- is less than zero, then the bodies are accelerating towards each other at the point and contact forces must be applied to prevent this acceleration. Once the forces are applied, the resulting acceleration a_{rel}^+ should be equal to or greater than zero. The applied forces have to follow several rules. They must be repellent and can only push objects away from each other and should not push them together. In other words, the magnitude of the contact force f has to be zero or positive. Additionally, once the relative acceleration becomes nonnegative, the contact forces must become zero, since forces are no longer acting once contact is broken. These constraints can be expressed as the set of inequalities:

$$a_{rel}^+ \geq 0, \quad f \geq 0, \quad a_{rel}^+ f = 0.$$

This is known as a complementarity condition where a^+_{rel} is complementary to f. Dealing with a single resting contact point is a simple problem. However, as Figure 4 shows, multiple bodies with simultaneous contact points are more difficult to resolve. The force magnitudes for each contact point have to be solved at once with each force and acceleration subject to complementarity constraints. The effect on each acceleration will be a linear combination of all the forces,

$$
\begin{gathered}
k_{11}f_1 + k_{12}f_2 + \cdots + k_{1n}f_n + a^-_{rel_1} = a^+_{rel_1} \, , \\
k_{21}f_1 + k_{22}f_2 + \cdots + k_{2n}f_n + a^-_{rel_2} = a^+_{rel_2} \, , \\
\vdots \\
k_{n1}f_1 + k_{n2}f_2 + \cdots + k_{nn}f_n + a^-_{rel_n} = a^+_{rel_n} \, .
\end{gathered}
$$

In matrix form, this can be rewritten as

$$
\mathbf{Kf} + \mathbf{a}^-_{rel} = \mathbf{a}^+_{rel},
$$

with $\mathbf{f}$ complementary to $\mathbf{a}^+_{rel}$. Such a system is referred to as a Linear Complementarity Problem (LCP).

A number of LCP solvers are available with particular strengths and weaknesses [14]. Whatever the solver used, the system will have the form

$$
\mathbf{Mz} + \mathbf{q} = \mathbf{w},
$$

where $\mathbf{z}$, $\mathbf{q}$, and $\mathbf{w}$ are vectors of n length and $\mathbf{M}$ is a matrix of size $n \times n$. $\mathbf{z}$ and $\mathbf{w}$ are the complementary variables, with the components of vectors being pairwise complementary. In terms of resting contact, $\mathbf{z}$ will be the vector of force magnitudes $\mathbf{f}$, $\mathbf{q}$ will be the initial accelerations $\mathbf{a}^-_{rel}$, and $\mathbf{w}$ will be the vector of resulting accelerations $\mathbf{a}^+_{rel}$. The inputs provided to the solver are the matrix $\mathbf{M}$ and the vector $\mathbf{q}$. The solver gives $\mathbf{z}$ and $\mathbf{w}$.

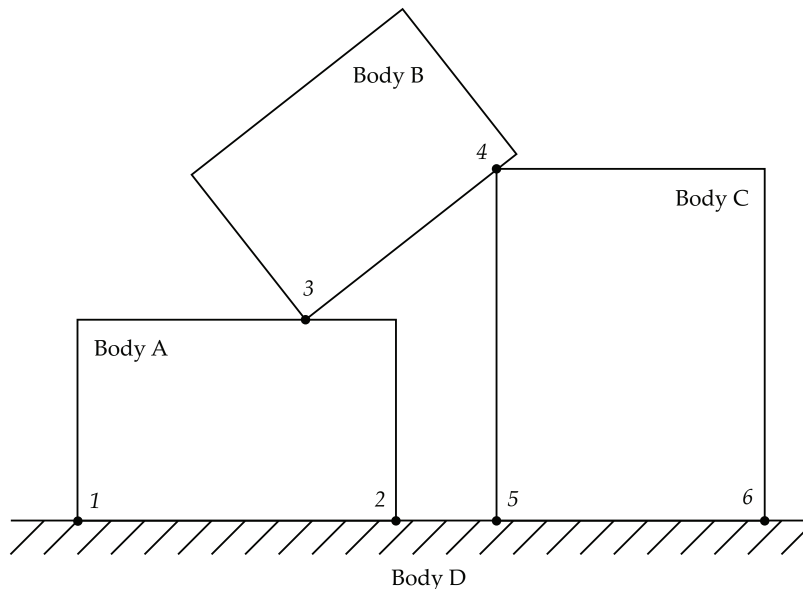

Fig. 4. An example of multiple resting contacts.
The contact points are numbered one through six with body D being static. Note that a force on any one contact affects at least one other contact. The system has to be solved simultaneously.

The vector **q** can be thought of as the initial state of the system and is populated by the relative accelerations of each contact point. However, computing the matrix **M** is not a trivial task. Essentially, m_{ij} is the effect that contact i has on contact j. If the two contacts are on two completely separate bodies, the coefficient will be zero. If the contacts have a common body, then their effect will be non-zero. Eberly [15] sets up this matrix by using mass properties m and **I** along with the collision normal **n** and the vector to the point of contact **r**.

B. Articulated Dynamics

While rigid body dynamics go a long way to solving particular problems in physics, they do not provide a complete description of constrained systems. For example,

modeling a door hinge or a drive train as a purely rigid body problem is quite complex. Having various objects in constant sliding, twisting, or other contact is very costly to simulate. Therefore, methods for handling the dynamics of constraints have to be considered for efficient simulation of such physical systems. The idea behind constrained dynamics is that instead of simulating contact constraints with collisions and resting contact, the constraints are part of the simulation state. The primary method described here for simulating constrained articulated structures is Featherstone's Articulated Body Method [8, 9, 5].

1. Articulated Structures

An articulated structure is a collection of joints and links in a tree-like arrangement. The joints act as connectors between rigid links of the structure. The simplest type of articulated structure is a chain with a fixed base as shown in Figure 5. Each joint in a chain can have only one parent and one child, and consequently only one joint on each end. We will build up the foundation for a chain first and then extend it to more complicated structures.

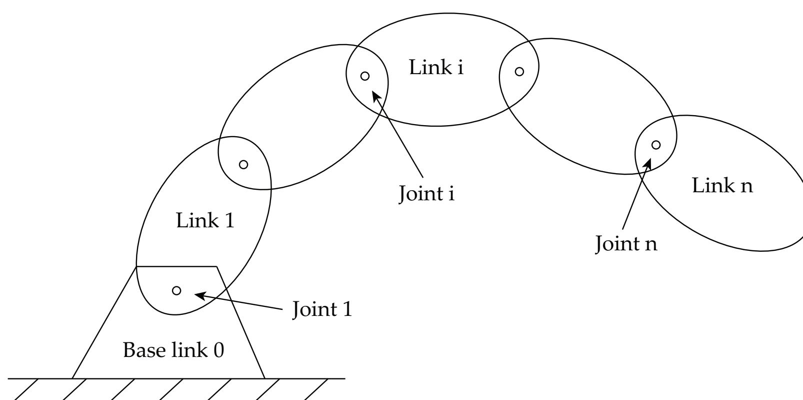

Fig. 5. An articulated chain with a fixed base.
This figure shows the conventions for joint and link ordering.

An articulated chain consists of n links and n joints (Figure 5). The links and joints are numbered from the base (inbound side) to the free end (outbound side) such that the parent of link i is link $i-1$. Each link i has corresponding inbound joint i. Each joint is either revolute or prismatic. Revolute joints rotate about a three-dimensional axis and prismatic joints slide along a three-dimensional joint axis (Figure 6). An additional requirement of the Featherstone algorithm is that each joint is limited to a single degree of freedom (DOF). This may sound very limiting, but more complicated multi-DOF joints can be created by placing two or more joints on top of each other with zero-mass links. An articulated structure will therefore have as many degrees of freedom as it has joints.

Fig. 6. Revolute and prismatic joints.

The Featherstone algorithm utilizes this representation to store the state of an articulated structure very compactly. For each joint, the only variables that are simulated are the joint angle and rate of change of the angle. The term “joint angle” is generalized. The angle is a measure of rotation in radians for revolute joints and a measure of displacement in world units for prismatic joints. The angle of joint i is denoted as q_i, its rate of change as velocity $\dot{q}_i$, and its second rate of change as acceleration $\ddot{q}_i$. Additionally, we will consider the joint torque Q_i as an actuator torque acting internally. The state for a single articulated structure can be determined by

the collection of these angles, their velocities, accelerations, and torques. These are designated as vectors $\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$, and $\mathbf{Q}$. The i_{th} component of each vector corresponds to the i_{th} joint. We start with the angles $\mathbf{q}$ and the velocities $\dot{\mathbf{q}}$ as the initial state. We must then determine the accelerations $\ddot{\mathbf{q}}$ acting on the system due to external forces and actuator torques $\mathbf{Q}$. The Articulated Body Method gives us these accelerations.

2. Joint and Link Properties

Before providing the algorithm for deriving the accelerations, we must establish a number of properties for the links and joints. We can treat the links as rigid bodies and therefore establish a coordinate system associated with each link. As we define properties, a subscript of i will denote that the property is in the coordinate system of that link unless otherwise noted.

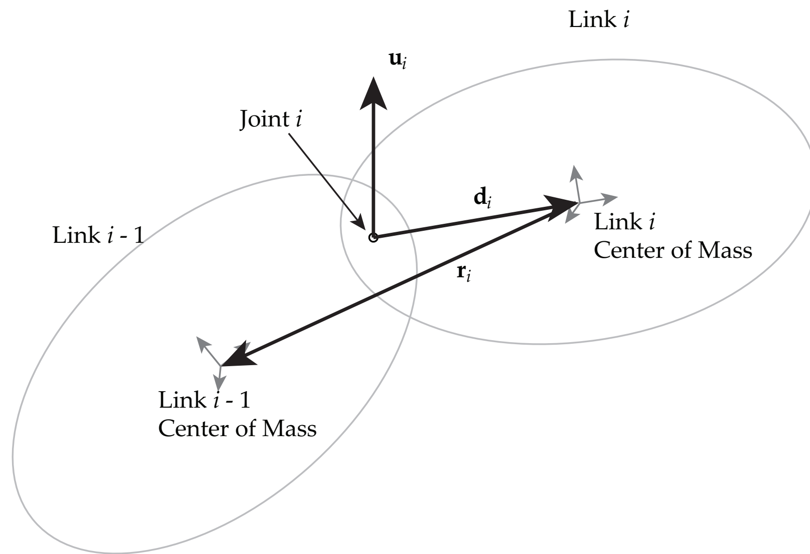

Fig. 7. Common vectors defined between links and joints.

Figure 7 shows several vectors we will need between links and joints. We define the joint axis as a three-dimensional vector $\mathbf{u}_i$. For revolute joints, this vector points

along the axis of rotation, with positive rotation defined by the right-hand rule. For prismatic joints, the vector is the direction of sliding. Since joint i moves and rotates with link i, the joint vector is constant in the coordinate frame of the link. The distance from the inbound joint to the center of mass of the link is denoted by $\mathbf{d}_i$. For revolute joints, this vector needs to be computed only once. For prismatic joints, this vector may change during simulation. Another vector $\mathbf{r}_i$ points from the center of the coordinate system of i to the center of the parent's coordinate system, $i-1$.

Each link will have an associated linear velocity $\mathbf{v}_i$ and angular velocity $\boldsymbol{\omega}_i$, as well as linear acceleration $\mathbf{a}_i$ and angular acceleration $\boldsymbol{\alpha}_i$. These properties describe the motion of link i in world space converted to the coordinate frame of i. It is convenient to define an additional pair of velocity variables: $\mathbf{v}_{rel}$ and $\boldsymbol{\omega}_{rel}$, which will be used in the derivations of velocity propagation along the chain, but will not be used in the actual algorithm. They denote the motion of the link itself due to its own joint velocity. Each link has mass m_i, an inertial tensor $\mathbf{I}_i$ defined in link coordinates, as well as matricized mass

$$\mathbf{M}_i = \begin{bmatrix} m_i & 0 & 0 \\ 0 & m_i & 0 \\ 0 & 0 & m_i \end{bmatrix}.$$

Note that although the links have position and rotation associated with them, these variables directly depend on the joint angles of the articulated structure.

We present here an abbreviated description of the velocity and acceleration derivations. For complete derivations, we direct the reader to the works of Mirtich and Featherstone [9, 8]. Basically, the only state information that we have are the joint angles and their velocities. We need to calculate the linear and angular velocities of the links from the state. In order to do that, we can divide the motion

of a link into two parts: one due to the motion of the link's parent, and one due to the motion of the link itself (i.e. the relative velocity). The velocity of link i is then

$$\boldsymbol{\omega}_i = \boldsymbol{\omega}_{i-1} + \boldsymbol{\omega}_{rel} \,,$$
$$\mathbf{v}_i = \mathbf{v}_{i-1} + \boldsymbol{\omega}_{i-1} \times \mathbf{r}_i + \mathbf{v}_{rel} \,.$$

The acceleration of link i is

$$\boldsymbol{\alpha}_i = \boldsymbol{\alpha}_{i-1} + \dot{\boldsymbol{\omega}}_{rel} \,,$$
$$\mathbf{a}_i = \mathbf{a}_{i-1} + \mathbf{a}_{i-1} \times \mathbf{r}_i + \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{r}_i) + \boldsymbol{\omega}_{i-1} \times \mathbf{v}_{rel} + \dot{\mathbf{v}}_{rel} \,.$$

The extra terms in the linear acceleration result from $\dot{\mathbf{r}}_i$. Using these equations, we can determine the velocities and accelerations of the entire chain if we know $\mathbf{v}_{rel}$ and $\boldsymbol{\omega}_{rel}$.

We will define two vectors

$$\boldsymbol{\nu}_i = \dot{q}_i \mathbf{u}_i \,,$$
$$\boldsymbol{\xi}_i = \ddot{q}_i \mathbf{u}_i \,,$$

to assist in the derivations of the relative properties. These vectors point along the joint axis and represent rotational velocity and acceleration if the joint is revolute or linear velocity and acceleration if the joint is prismatic. For prismatic joints, the relative properties are

$$\boldsymbol{\omega}_{rel} = \mathbf{0} \,,$$
$$\dot{\boldsymbol{\omega}}_{rel} = \mathbf{0} \,,$$
$$\mathbf{v}_{rel} = \boldsymbol{\nu}_i \,,$$
$$\dot{\mathbf{v}}_{rel} = \boldsymbol{\xi}_i + \boldsymbol{\omega}_{i-1} \times \boldsymbol{\nu}_i \,,$$

and for revolute joints

$$\begin{aligned}
\boldsymbol{\omega}_{rel} &= \boldsymbol{\nu}_i \,, \\
\dot{\boldsymbol{\omega}}_{rel} &= \boldsymbol{\xi}_i + \boldsymbol{\omega}_{i-1} \times \boldsymbol{\nu}_i \,, \\
\mathbf{v}_{rel} &= \boldsymbol{\nu}_i \times \mathbf{d}_i \,, \\
\dot{\mathbf{v}}_{rel} &= \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) + \boldsymbol{\xi}_i \times \mathbf{d}_i + \boldsymbol{\nu}_i \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) \,.
\end{aligned}$$

Since the algorithm deals with these rather large equations, a compact form for representing them is necessary.

3. Spatial Algebra

Spatial algebra is a notation tool developed for efficiently describing three-dimensional quantities. A spatial vector is a six-dimensional vector that encompasses two three-dimensional vectors. For example, it is possible to describe velocity by a single vector that contains both the linear and angular velocity. A spatial matrix becomes a 6×6 matrix containing four 3×3 sub-matrices. A spatial vector is denoted by a hat symbol over the variable. Spatial velocity and accelerations are therefore described as:

$$\hat{\mathbf{v}} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} , \quad \hat{\mathbf{a}} = \begin{bmatrix} \boldsymbol{\alpha} \\ \mathbf{a} \end{bmatrix} .$$

Note that the rotational component is at the top of the spatial vector and the linear component is at the bottom.

Traversing an articulated structure requires transforming spatial vectors from one link's coordinate frame to another. To convert a spatial vector $\hat{\mathbf{v}}_F$ in coordinate frame F to $\hat{\mathbf{v}}_G$ coordinate frame G, we will construct a spatial transformation matrix ${}_G\hat{\mathbf{X}}_F$ so that $\hat{\mathbf{v}}_G = {}_G\hat{\mathbf{X}}_F\hat{\mathbf{v}}_F$.

If $\mathbf{r}$ is the offset from F to G expressed in G's coordinate frame, we define the

following operator on $\mathbf{r}$:

$$\mathbf{r}\times = \begin{bmatrix} 0 & -\mathbf{r}_z & \mathbf{r}_y \\ \mathbf{r}_z & 0 & -\mathbf{r}_x \\ -\mathbf{r}_y & \mathbf{r}_x & 0 \end{bmatrix},$$

which represents a cross product with its forming vector. Premultiplying a vector by this matrix is the equivalent of taking the cross product of $\mathbf{r}$ and that vector. Thus, if the coordinate transformation were purely translational, the transform from F to G would be

$${}_G\hat{\mathbf{T}}_F = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{r}\times & \mathbf{1} \end{bmatrix},$$

where the $\mathbf{1}$ is the 3×3 identity matrix and $\mathbf{0}$ is the 3×3 zero matrix. This transformation can be derived

$$\hat{\mathbf{v}}_G = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_G \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_F + \mathbf{r} \times \boldsymbol{\omega} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{r}\times & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_F \end{bmatrix},$$

which does not change angular velocity but does change the linear velocity.

We can construct a 3×3 transformation matrix ${}_G\mathbf{R}_F$ for pure rotation from frame F to frame G. A rotation will affect both angular and linear components, so we can construct a spatial rotation transformation matrix as follows:

$${}_G\hat{\mathbf{R}}_F = \begin{bmatrix} {}_G\mathbf{R}_F & \mathbf{0} \\ \mathbf{0} & {}_G\mathbf{R}_F \end{bmatrix}.$$

Thus, the full spatial transformation matrix containing both translation and rotation is

$${}_G\hat{\mathbf{X}}_F = \begin{bmatrix} {}_G\mathbf{R}_F & \mathbf{0} \\ \mathbf{0} & {}_G\mathbf{R}_F \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{r}\times & \mathbf{1} \end{bmatrix} = \begin{bmatrix} {}_G\mathbf{R}_F & \mathbf{0} \\ \mathbf{r} \times {}_G\mathbf{R}_F & {}_G\mathbf{R}_F \end{bmatrix}.$$

We can also define a spatial cross product operator $\hat{\times}$ on spatial vectors. Assum-

ing that

$$\hat{\mathbf{x}} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix},$$

we define $\hat{\mathbf{x}}\hat{\times}$ as

$$\hat{\mathbf{x}}\hat{\times} = \begin{bmatrix} \mathbf{a}\times & \mathbf{0} \\ \mathbf{b}\times & \mathbf{a}\times \end{bmatrix}.$$

To take the spatial inner product of two vectors, we need the spatial transpose operator $'$, defined by

$$\hat{\mathbf{x}}' = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}' = \begin{bmatrix} \mathbf{b}^T & \mathbf{a}^T \end{bmatrix}.$$

Using this operator, we can obtain useful properties such as power from spatial vectors. Assume that a spatial force vector is defined

$$\hat{\mathbf{f}} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} \end{bmatrix}.$$

The spatial inner product between spatial force and spatial velocity is

$$\hat{\mathbf{f}}'\hat{\mathbf{v}} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} \end{bmatrix}' \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^T & \boldsymbol{\tau}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} = \boldsymbol{\tau}\cdot\boldsymbol{\omega} + \mathbf{f}\cdot\mathbf{v},$$

which is the definition of power.

Note that the spatial force vector is defined with the angular component on the bottom and the linear on the top. The translational spatial transformation will act correctly and modify the torque, but not the force. For an applied force that produces a linear force and torque, this makes sense since offsetting the acting point will produce a different torque but the same linear force.

In spatial algebra, the spatial joint axis can be represented as

$$\hat{\mathbf{s}} = \begin{bmatrix} \mathbf{0} \\ \mathbf{u}_i \end{bmatrix}$$

for prismatic joints and

$$\hat{\mathbf{s}} = \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_i \times \mathbf{d}_i \end{bmatrix}$$

for revolute joints. This vector remains constant in body coordinates so it only needs to be computed once. One great advantage of spatial notation already manifests itself: both revolute and prismatic joints can be represented with the same equations (the components of course are defined differently).

When deriving acceleration, several extra terms show up that depend on the velocity of the moving coordinate frame and are known as Coriolis forces. The spatial Coriolis force of link i is

$$\hat{\mathbf{c}}_i = \begin{bmatrix} 0 \\ \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{r}_i) + 2\boldsymbol{\omega}_{i-1} \times \boldsymbol{\nu}_i \end{bmatrix}$$

if link i is connected to a prismatic joint and

$$\hat{\mathbf{c}}_i = \begin{bmatrix} \boldsymbol{\omega}_{i-1} \times \boldsymbol{\nu}_i \\ \boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times \mathbf{r}_i) + 2\boldsymbol{\omega}_{i-1} \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) + \boldsymbol{\nu}_i \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) \end{bmatrix}$$

if connected to a revolute joint.

Revisiting the acceleration equations, the acceleration of a link depends on the parent's acceleration and the change in relative velocity of that link. The change in relative velocity depends on two components: $\ddot{q}_i$ and the Coriolis components. We now have the means of expressing acceleration propagation from parent to child as

$$\hat{\mathbf{a}}_i = {}_i\hat{\mathbf{X}}_{i-1}\hat{\mathbf{a}}_{i-1} + \ddot{q}_i\hat{\mathbf{s}}_i + \hat{\mathbf{c}}_i.$$

This applies to both revolute and prismatic joints. Additionally, we can express the velocity as

$$\hat{\mathbf{v}}_i = {}_i\hat{\mathbf{X}}_{i-1}\hat{\mathbf{v}}_{i-1} + \dot{q}_i\hat{\mathbf{s}}_i.$$

4. The Featherstone Articulated Body Method

With the velocity and acceleration propagation defined, the algorithm for determining the accelerations of the entire articulated structure can be described. The algorithm works by successively solving sub-chains of the original structure starting with the trivial case of the last link. This process is very well documented by Mirtich. We will present an abbreviated version of this proof.

The motion of a link in an articulated chain is determined by the forces and torques acting upon it. The center of mass is affected by gravity, external forces, the angular velocity of the body, and the forces and torques exerted by inbound $(\mathbf{f}_i^I, \boldsymbol{\tau}_i^I)$ and outbound $(\mathbf{f}_i^O, \boldsymbol{\tau}_i^O)$ joints. Considering the last joint n in the link, we have

$$\begin{aligned} \mathbf{f}_n^I + m_n\mathbf{g} &= m_n\mathbf{a}_n\,, \\ \boldsymbol{\tau}_n^I &= \mathbf{I}_n\boldsymbol{\alpha}_n + \boldsymbol{\omega}_n \times \mathbf{I}_n\boldsymbol{\omega}_n\,. \end{aligned}$$

We can express this pair of equations as one spatial equation using matricized mass:

$$\begin{array}{ccccccc} \begin{bmatrix} \mathbf{f}_n^I \\ \boldsymbol{\tau}_n^I \end{bmatrix} & = & \begin{bmatrix} \mathbf{0} & \mathbf{M}_n \\ \mathbf{I}_n & \mathbf{0} \end{bmatrix} & \begin{bmatrix} \boldsymbol{\alpha}_n \\ \mathbf{a}_n \end{bmatrix} & + & \begin{bmatrix} -m_n\mathbf{g} \\ \boldsymbol{\omega}_n \times \mathbf{I}_n\boldsymbol{\omega}_n \end{bmatrix}, \\ \hat{\mathbf{f}}_n^I & = & \hat{\mathbf{I}}_n & \hat{\mathbf{a}}_n & + & \hat{\mathbf{Z}}_n\,. \end{array}$$

The general form of the previous equation gives us several new properties. The spatial acceleration has already been defined; however, the other three deserve some attention. The left-hand side is the inbound spatial force. It contains the force and torque exerted by the inbound joint. $\hat{\mathbf{I}}_i$ is the isolated spatial inertial tensor that contains the mass properties for this link. It is referred to as an isolated property

because in an articulated chain, the spatial inertial tensor will be accumulated to reflect the effect of the rest of the chain. The accumulated spatial inertial tensor is denoted as $\hat{\mathbf{I}}_i^A$. The accumulated spatial inertial tensor for the nth link is the same as the isolated property. Lastly, $\hat{\mathbf{Z}}_i$ is referred to as the isolated spatial zero-acceleration force, named so because without this vector, the link would not accelerate. Any additional external spatial forces acting on the object would also be part of this vector. Like the spatial inertial tensor, the zero-acceleration vector will be accumulated to reflect forces acting throughout the chain.

Once we know the relationship between the force and accelerations of the nth link, we can inductively derive a relationship for the rest of the chain. The $i-1$ link has the same formula as the last link with the addition of the force exerted by the outbound joint,

$$\hat{\mathbf{f}}_{i-1}^I = \hat{\mathbf{I}}_{i-1}\hat{\mathbf{a}}_{i-1} + \hat{\mathbf{Z}}_{i-1} - \hat{\mathbf{f}}_{i-1}^O.$$

The force of the inbound joint is equal and opposite to the force of the outbound joint force,

$$\hat{\mathbf{f}}_{i-1}^O = -{}_{i-1}\hat{\mathbf{X}}_i\hat{\mathbf{f}}_i^I.$$

Thus,

$$\hat{\mathbf{f}}_{i-1}^I = \hat{\mathbf{I}}_{i-1}\hat{\mathbf{a}}_{i-1} + \hat{\mathbf{Z}}_{i-1} - {}_{i-1}\hat{\mathbf{X}}_i(\hat{\mathbf{I}}_i^A\hat{\mathbf{a}}_i + \hat{\mathbf{Z}}_i^A).$$

Note that the spatial inertial tensor and the zero-acceleration vector for the ith link is the accumulated form. In order to solve this equation, the acceleration of link i must be expressed in terms of link $i-1$. A detailed derivation is presented by Mirtich [9]. He shows that

$$\begin{aligned}
\hat{\mathbf{I}}_{i-1}^A &= \hat{\mathbf{I}}_{i-1}^A + {}_{i-1}\hat{\mathbf{X}}_i\left(\hat{\mathbf{I}}_i^A - \frac{\hat{\mathbf{I}}_i^A\hat{\mathbf{s}}_i\hat{\mathbf{s}}_i'\hat{\mathbf{I}}_i^A}{\hat{\mathbf{s}}_i'\hat{\mathbf{I}}_i^A\hat{\mathbf{s}}_i}\right){}_i\hat{\mathbf{X}}_{i-1}\,, \\
\hat{\mathbf{Z}}_{i-1}^A &= \hat{\mathbf{Z}}_{i-1}^A + {}_{i-1}\hat{\mathbf{X}}_i\left(\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A\hat{\mathbf{c}}_i + \frac{\hat{\mathbf{I}}_i^A\hat{\mathbf{s}}_i\left[Q_i - \hat{\mathbf{s}}_i'\left(\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A\hat{\mathbf{c}}_i\right)\right]}{\hat{\mathbf{s}}_i'\hat{\mathbf{I}}_i^A\hat{\mathbf{s}}_i}\right).
\end{aligned}$$

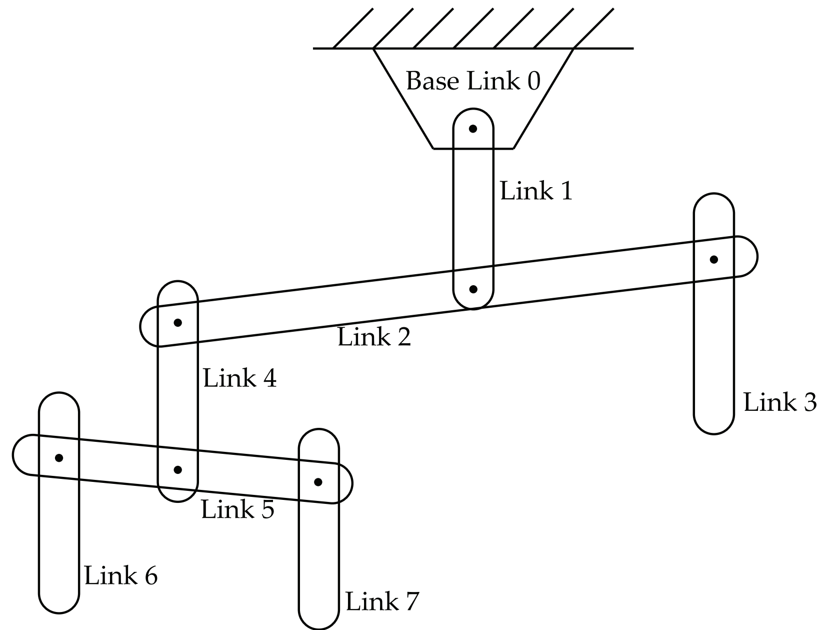

Fig. 8. A tree-like articulated structure.
Note the way the joints are numbered: a traversal in either direction ensures that all the information is available.

The extension to tree-like linkages is rather simple. For chains, the parent of link i is link $i-1$. For tree-like articulated structures, the parent $i-1$ becomes another link denoted by the subscript h such that $h < i$. This ensures that the children of a link will have all the properties they need from the parent and that the parent will have all of the children's accumulated properties (Figure 8).

We now know how to compute all of the properties that we need to determine the joint and spatial accelerations of an articulated body. The algorithm itself works in three loops as shown in Figure 9. The first loop is outbound from the root to the children and computes the velocity, Coriolis forces, isolated inertial tensors, and zero-acceleration vectors. The second loop is inbound and computes all of the accumulated properties. The third loop is again outbound and yields the spatial and joint accelerations of each link. In order to avoid performing the same calculations

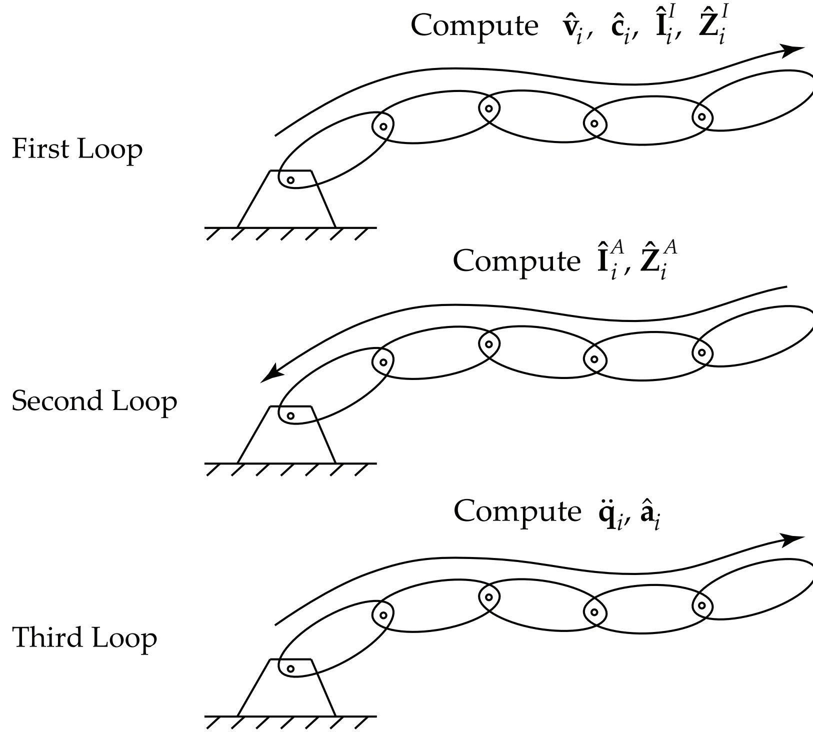

Fig. 9. Articulated body method.

multiple times, a number of common properties are defined. These are

$$\begin{aligned}
\hat{\mathbf{h}}_i &= \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i \,, \\
d_i &= \hat{\mathbf{s}}_i' \hat{\mathbf{h}}_i \,, \\
u_i &= Q_i - \hat{\mathbf{h}}_i \hat{\mathbf{c}}_i - \hat{\mathbf{s}}_i' \hat{\mathbf{Z}}_i^A \,.
\end{aligned}$$

Using these definitions, Kokkevis [5] presents a compact version of the algorithm. The pseudocode for the Articulated Body Method is presented in Algorithm 1. All variables are in the coordinate space of their subscript.

The algorithm is fairly straight-forward, but some things deserve a bit more explanation. When initializing the zero-acceleration vector in the first loop, we use a spatial force vector $\hat{\mathbf{f}}_i^E$ that we get from $\mathbf{F}^E$, that is the sum of all external forces

Algorithm 1 Pseudocode for the Articulated Body Method.

procedure ABMACCELERATIONS($\mathbf{q}$, $\dot{\mathbf{q}}$, $\mathbf{F}^E$, $\mathbf{Q}$)

  $\hat{\mathbf{v}}_0 = \hat{\mathbf{0}}$

  // First outbound loop

  for $i = 1$ to n **do**

    h = index of parent of link i

    $\hat{\mathbf{v}}_i = {}_i\hat{\mathbf{X}}_h \hat{\mathbf{v}}_h + \dot{q}_i \hat{\mathbf{s}}_i$

    $\hat{\mathbf{I}}_i^A = \hat{\mathbf{I}}_i$

    $\hat{\mathbf{Z}}_i^A = \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{I}}_i^A - \hat{\mathbf{f}}_i^E$

    $\hat{\mathbf{c}}_i = \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{s}}_i \dot{q}_i$

  // Inbound loop

  for $h = n$ to 1 **do**

    if link h has a child **then**

      i = index of child of link h

      $\hat{\mathbf{I}}_h^A = \hat{\mathbf{I}}_h^A + {}_h\hat{\mathbf{X}}_i \left(\hat{\mathbf{I}}_i^A - \frac{\hat{\mathbf{h}}_i \hat{\mathbf{h}}_i'}{d_i} \right) {}_i\hat{\mathbf{X}}_h$

      $\hat{\mathbf{Z}}_h^A = \hat{\mathbf{Z}}_h^A + {}_h\hat{\mathbf{X}}_i \left(\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i + \frac{u_i}{d_i} \hat{\mathbf{h}}_i \right)$

    $\hat{\mathbf{h}}_h = \hat{\mathbf{I}}_h^A \hat{\mathbf{s}}_h$

    $d_h = \hat{\mathbf{s}}_h' \hat{\mathbf{h}}_h$

    $u_h = Q_h - \hat{\mathbf{h}}_h' \hat{\mathbf{c}}_h - \hat{\mathbf{s}}_h' \hat{\mathbf{Z}}_h^A$

  // Second outbound loop

  $\hat{\mathbf{a}}_0 = \hat{\mathbf{0}}$

  for $i = 1$ to n **do**

    h = parent of link i

    $\ddot{q}_i = \frac{u_i - \hat{\mathbf{h}}_i' {}_i\hat{\mathbf{X}}_h \hat{\mathbf{a}}_h}{d_i}$

    $\hat{\mathbf{a}}_i = {}_i\hat{\mathbf{X}}_h \hat{\mathbf{a}}_h + \hat{\mathbf{c}}_i + \hat{\mathbf{s}}_i \ddot{q}_i$

acting on each link. This can include things such as gravity, contact forces, and user applied forces. If the forces are applied in the world coordinate space they must be converted into the appropriate link coordinate space.

5. Articulated Dynamics State

The state of an articulated chain is given by the state vector

$$\mathbf{X} = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix},$$

which records both the joint angles and their velocities. The length of $\mathbf{q}$ and $\dot{\mathbf{q}}$ is determined by the number of joints or degrees of freedom in the chain. The derivative of the state,

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix},$$

requires the accelerations, which are given to us by the Articulated Body Method, and include the effects of joint torques $\mathbf{Q}$ and external forces $\hat{\mathbf{f}}_i^E$.

While that is all of the state information that we need for integration, the data structures for articulated links will carry a number of extra variables with them. Spatial velocities, accelerations, and mass properties are needed for the Articulated Body Method as well as transformation information for the algorithm as well as display purposes.

6. Collision Response

The collision response for articulated bodies follows similar guidelines as those for rigid bodies. Resolving colliding contacts requires impulses that will change v_{rel}^- to the desired v_{rel}^+. Similarly, for resting contacts, a Linear Complementarity Problem is set

up with a complementarity condition between the forces and accelerations. However, determining impulse magnitudes and the matrix for the Linear Complementarity Problem is more involved. Additionally, impulses and forces have to be propagated throughout the entire chain.

The problem with the impulse approach described in the section III.A.3 is that links in articulated structures do not have easily defined inertial tensors and scalar masses. Thankfully, it is not necessary to derive 3×3 inertial tensors and scalar masses from their articulated spatial counterparts. The expression for the magnitude of the impulse for rigid bodies shows how the two bodies will react to a test impulse. In order to solve the same problem with articulated links, the simplest approach is to use test impulses and simulate their effects instead of calculating the magnitude directly.

The Featherstone algorithm gives a way of applying an arbitrary force to any of the links by setting the appropriate link's applied force $\mathbf{F}^E$. This can be used for testing the response to impulses. The effect of an impulse j will change pre-impulse joint velocities $\dot{\mathbf{q}}^-$ to post-impulse joint velocities $\dot{\mathbf{q}}^+$. We will do this by propagating a force of the same magnitude as the impulse throughout the articulated body and recording the change in accelerations $\ddot{\mathbf{q}}^I$. The resulting joint velocities,

$$\dot{\mathbf{q}}^+ = \dot{\mathbf{q}}^- + \ddot{\mathbf{q}}^I \delta t\,,$$

are obtained by adding the change in accelerations to the pre-impulse joint velocities. We can set $\delta t = 1$ because applying an impulse $j = f \delta t$ to an object's momentum produces the same effect as applying a force f and integrating it over a timestep δt. Since we are not performing an integration step, the choice of δt becomes irrelevant. The benefit that we get from using a test impulse is that we can determine how the articulated body reacts to it without having to explicitly calculate inertial and mass

properties.

For a single contact, we can apply a unit test impulse acting in the direction of the contact normal to the link of the articulated structure. There is a linear relationship between the force applied and the change in accelerations and therefore the change in velocities [5]. This linear relationship between the force and velocity can be represented as a scalar

$$k = v_{rel}^{t} - v_{rel}^{-}$$

where v_{rel}^{t} is the resulting relative velocity after the test impulse. Thus, the magnitude of the actual impulse needed to achieve the desired relative velocity is

$$f = \frac{v_{rel}^{+} - v_{rel}^{-}}{k}.$$

An actual impulse of magnitude f is applied to the articulated chain to update all velocities.

The resting contact problem is solved in a similar manner as the rigid body problem. The same equations are set up with the matrix built using test impulses. Once the forces are computed by the LCP solver, all the forces have to be applied at once to the entire articulated structure.

C. Immersive Engine

The Visualization Laboratory at Texas A&M University holds a relatively new immersive system [3]. The system is comprised of a number of computers, projectors, and modular screens. Each computer is powered by an Intel processor and uses the same type of video card. The video output is fed to a set of projectors that place an image on a set of screens. The screens surround the user in some configuration, such as an approximation to a sphere or a cylinder (Figure 10).

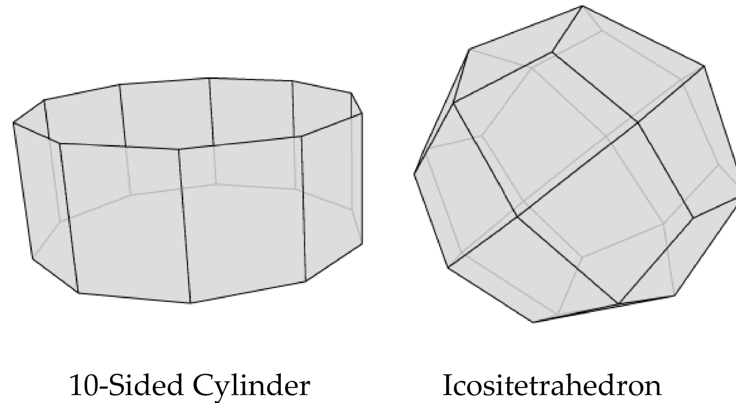

Fig. 10. Example cave geometries.
Current installations use 4 or 5 faces of the full geometry.

A single machine is designated as the server and the rest as the clients. The software that runs this system is known as GUPPY3D [16]. It is responsible for loading scene data including models, textures, and lights. Additionally, the software ensures that the server and clients communicate appropriately and do not get out of sync. A cave configuration file is loaded at startup, providing information about the physical location of the screens. GUPPY3D allows each computer to be configured to a particular screen, letting the user adjust screen rotation, orientation, and correct for perspective and other distortions. When properly adjusted, GUPPY3D displays a single image over multiple facets providing the user with an immersive visual experience. The system relays user and camera position and orientation between the server and clients via a network, allowing the user to move through the virtual world with all the screens maintaining one coherent scene.

The user interacts with the server machine using a keyboard and mouse. The keyboard is used for moving the user through the world by changing his position and elevation. The mouse controls the user's orientation, allowing exploration of the virtual world.

While the system presents the user with immersive visuals, the interaction is limited to just viewing the world. There is some ability to script the motion of objects or trigger object motion. However, there is currently no dynamic behavior present in the system.

CHAPTER IV

METHODOLOGY AND IMPLEMENTATION

This chapter focuses on the design methodology behind the physics engine and the implementation. The physics engine is designed as a very modular and easily expandable system; the chapter will detail the ideas behind the strong object-oriented design and liberal use of interfaces to achieve modularity. For clarity, we will need to differentiate between several "engines" present in the system. The first is the immersive rendering engine, known as GUPPY3D. We will refer to it as the *immersive engine*. The next is the engine specifically developed for the dynamics. We will refer to that as the *physics engine*. Lastly, the main class in the physics engine is also called Engine and will be referred to as the `Engine` class.

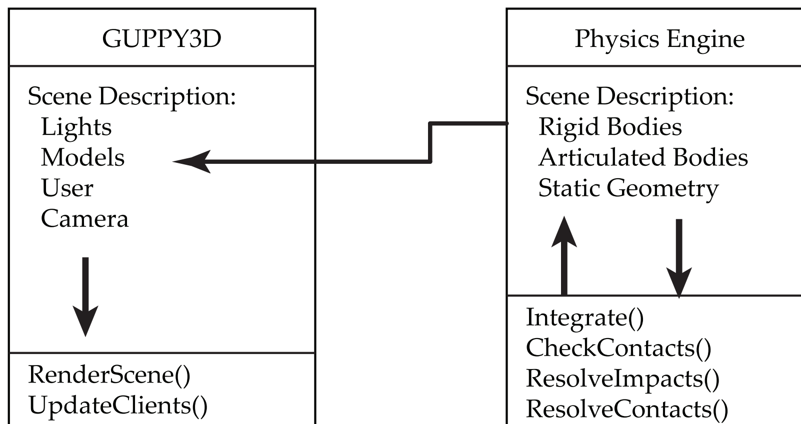

Fig. 11. GUPPY3D and the physics engine.
The rendering engine GUPPY3D is responsible for rendering the scene information. The physics engine simulates the models from the immersive scene and updates their properties.

The primary goal behind the physics engine is to provide simulation support for the immersive rendering environment. As seen in Figure 11, the physics engine works with the immersive engine by controlling various objects present in the system. Those

objects are simulated in the physics engine and their positions and orientations are updated. The immersive engine continues its display loop and renders the objects.

Figure 12 shows object hierarchy and object relationships to the interfaces. It shows that most classes that influence the behavior of the dynamics inherit from a base `Object` class as well as a number of interfaces. The interfaces enforce critical functionality and allow these objects to be used in the simulation system, in rendering, and in collision detection.

A. Interfaces

The best way to understand what functionality objects have is by the interfaces that they can implement. The interfaces that are critical to the physics engine are `SimulationState`, `Integratable`, `Collidable`, `Constraint`, `NetSync`, and `SidRenderable`. Other interfaces `Renderable`, `Camera`, and `Light` can be used if the framework is used in a standalone environment. We will outline the functionality of the interfaces and explain how they are used by the physics engine.

1. `SimulationState`

A class implementing the `SimulationState` interface will usually be very specialized and will primarily deal with storing the state of a dynamic object. In the case of rigid bodies, this class will have three-dimensional vectors for position, linear momentum, angular momentum, and a quaternion for rotation. It will also have pointers to auxiliary properties that are not part of the state but are required for certain operations. These properties include inertial tensors, velocities, forces, torques, and rotation matrices. For articulated dynamics, a simulation state class will hold an array of joint angles and joint velocities. It will also have pointers to joint accelerations, joint

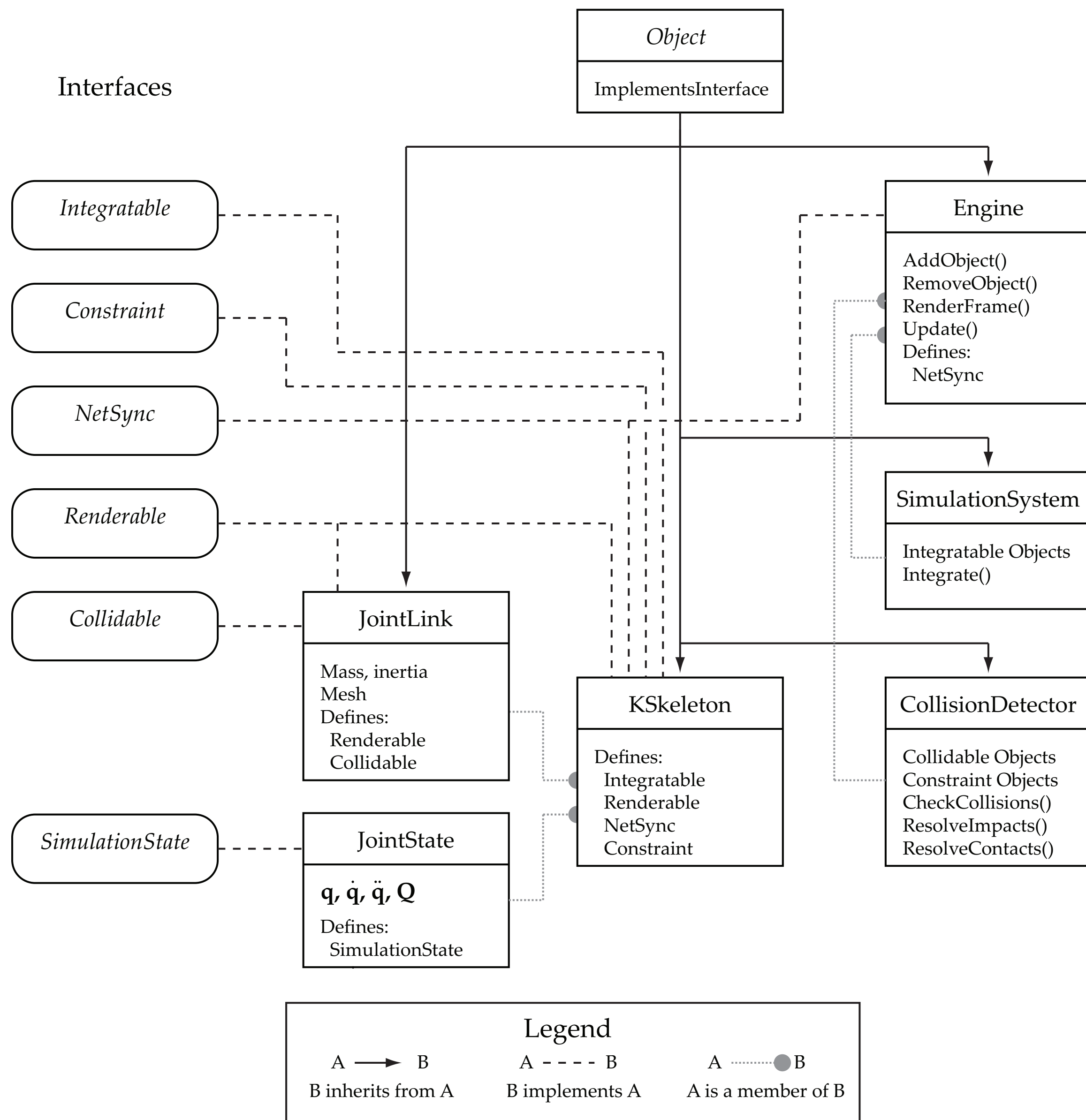

Fig. 12. Object hierarchy and interface diagram.

An example of how the kinematic skeleton `KSkeleton` fits into the class structure. The `KSkeleton` class is an integratable, renderable, synchronizable, constraint object that uses a `JointState` class to store its state information and `JointLink` objects for collision detection and rendering. The `Engine` class adds the skeleton and its objects to other objects based on the interfaces they implement.

torques, and pointers to the array of links.

A simulation state class must be able to do several things: the class has to define a set of self-modifying operators and the ability to compute the rate of change of the state from the current state and time. The derivative of the state must take into account the entire system. A simple Euler integration at time t with timestep h is

$$\mathbf{X}(t+h) = \mathbf{X}(t) + \dot{\mathbf{X}}(t)h\,,$$

which computes $\mathbf{X}(t+h)$, the state after the timestep, from the current state $\mathbf{X}(t)$ and its derivative $\dot{\mathbf{X}}(t)$. Essentially, we are defining the operators needed to perform this function. The derivative operator is rather simple and has been described for both rigid bodies and articulated dynamics. A state also has to be able to be multiplied by the timestep h and added to another state. This implies definitions for the multiplication `*` operator and the addition `+` operator and finally the assignment `=` operator. Once these are defined, a state class has all the functionality it needs to be integrated. The class definition for the `SimulationState` is shown below.

```
class SimulationState {
public:
  virtual void Derivative() = 0;
  virtual void UpdateAuxiliaryProperties() = 0;
  virtual SimulationState& operator = (const SimulationState& s) = 0;
  virtual SimulationState& operator += (const SimulationState &s) = 0;
  virtual SimulationState& operator *= (const double &h) = 0;
  virtual SimulationState& operator /= (const double &h) = 0;
};
```

Previously, we mentioned that the mathematical operators are self-modifying. Since the `SimulationState` interface is an abstract class, it cannot be instantiated. Reference parameters to `SimulationState` interfaces are therefore not possible and C++ requires that at least one of the operands in a binary operator is not a pointer. Additionally, a reference to a `SimulationState` interface must be returned, which again is not possible due to the abstract nature of the interface. Although some-

what cumbersome, by using a temporary state, we can get some speed out of this restriction. Note that in the Euler integration, there are three operators in effect: derivative, multiplication, and addition. Traditionally, each of these operators would create a temporary new state (applying the assignment operator as well). For articulated dynamics, this would mean allocating arrays of memory for all the joint angles and velocities. This would happen three times and the intermediate data would be discarded at the end. Using self-modifying operators, we can achieve the same result using an extra temporary state and one extra equate statement:

```
*tempState = *nextState = *currentState;
*tempState.Derivative();
*tempState *= h;
*nextState += *tempState;
```

This is not as clean as performing the whole operation in one line, but we avoid memory allocation and deallocation.

2. `Integratable`

The `Integratable` interface is used by the `SimulationSystem` class in the physics engine to advance objects by a timestep. Objects that implement the `Integratable` interface can be added to the simulation system. The simulation system will then ask each object to accumulate the forces affecting it, take the derivative of its state, and advance its state using Euler integration.

The `SimulationState` is the bread and butter class of the `Integratable` interface. An object that defines an `Integratable` interface automatically gains a set of pointers to the `SimulationState` interface and some basic integration functionality. The `Integratable` interface defines `SimulationState` interfaces for current state, next state, previous state, initial state, and temporary state. Any class inheriting from `Integratable` is guaranteed to have those states defined. The other critical

functionality is the ability to accumulate forces and update the state of the object. This functionality is shown in the `Integratable` interface:

```
class Integratable {
protected:
  SimulationState* currentState, previousState, nextState;
                   tempState, initialState;
  bool isStatic, isAlive, isIntegratable;
public:
  virtual void AccumulateForces() = 0;
  virtual void UpdateState(const SimulationStateState &state) = 0;
  virtual void Stabilization(const double &delta) = 0;
};
```

Force accumulation is necessary for taking the appropriate derivatives. At any one time, an object may have several forces acting on it. Whether those forces are gravity, wind, contact forces, or other constraint forces that may relate to other objects, all of them have to be accounted for in order to get the proper accelerations. The physics engine assumes that the accumulated force information is used to correctly determine the derivative. Each object itself does not necessarily need to know all the forces that are acting on it. Constraint objects, for example, can affect others by applying additional forces. For instance, a number of particles can be governed by a "springy mesh" object. The particles accumulate gravity forces by themselves and the springy mesh object provides additional spring forces. Each particle does not have to know that it is connected to other particles. The springy mesh object acts as a force generator and takes care of those forces in its own accumulate forces function. The `AccumulateForces()` call will be performed for each object before any derivatives are taken.

`UpdateState()` is the other critical function that an `Integratable` class must define. This method is called to insure that an object has all of its relevant information updated given a particular state. For rigid bodies, for example, this method would ensure that the linear and angular velocity are up to date, since it is linear and angular momentum that are stored in the state. During collision detection, object

velocities are used to determine the relative velocity of contact points. For articulated structures, the state consists of joint information only, so all of the velocities and accelerations must be updated. Note that the Articulated Body Method call would actually go here.

Lastly, the function `Stabilization()` is used to take care of any issues that may have to be resolved after integration. This function is frequently not necessary, but at times critical. For example, it could be used to threshold velocities for objects or find conservative values for grid-based velocity systems.

3. `Collidable`

The `Collidable` interface is designed to be used with the `CollisionDetector` class of the physics engine. During the simulation loop, the collision detector is responsible for several tasks. The detector needs to determine whether there are any object intersections. If there are object contacts, it needs to provide the specific contact information such as point, normal, and relative velocity. For resting contacts, it needs to know the relative acceleration of the contact point. The `Collidable` interface is used for performing such tasks. Because the code for the `Collidable` interface is rather long, we present it in several sections. The first section defines some members and virtual methods:

```
class Collidable {
protected:
  Vector3d contactForces, contactTorques;
  bool isCollidable;
  CollisionObjectType coType;
public:
  virtual int Intersect(Collidable *co) = 0;
  virtual bool CollidesWith(const CollisionObjectType &type) = 0;
```

Each `Collidable` class defines a type stored as an enum. This type is used to distinguish the various kinds of objects that may be in a dynamic scene. Some objects will naturally interact with others. Some pairings do not need any collision detection

or response to be performed between them. For example, planes and static meshes do not move and therefore do not need to be tested for collision. Each `Collidable` class, therefore, defines a `CollidesWith()` method that returns true if that class collides with a particular type. In addition, a method that returns the type of the object is also defined.

If an object does collide with another object, the `Intersect()` method is called. This method passes in the target `Collidable` interface object as a parameter. Based on the type of the passed `Collidable` interface, different methods of intersection testing are used. For example, collision tests between two rigid bodies and a rigid body and a plane will use different methods. The `Intersect()` method returns an integer that specifies how two objects intersect. A value of -1 means that the objects are completely disjoint. A value of 0 means that an exact contact or contacts exist between the two objects. When a contact occurs, a `Contact` object is added to the collision detector. Finally, the value +1 means that at least one intersection between the two objects has occurred. Note that even if contacts exist, an intersection overrides the contacts. The simulation loop, which will be discussed in greater detail later, discards contact information if an intersection is found and rolls back the simulation to the time of exact contact.

When the collision detector finds a valid set of contacts, it must resolve them. Contacts occur between objects implementing the `Collidable` interface, so all those objects must implement the necessary functionality for contact resolution. The following functions are necessary: getting the velocity of a point, the acceleration of a point, the object's position, the object's restitution coefficient, and the object's friction coefficient. These methods allow the collision detector to determine whether a contact is a colliding contact based on the relative velocity of the two points. If the relative velocity of a point is zero, the acceleration of a point is used to deter-

mine whether the contact is a resting contact. The position of the object relative to the point of contact determines the torque resulting from impulses and forces and the restitution and friction coefficients determine the magnitude of the impulses and forces. These methods form the next section of the `Collidable` interface:

```
virtual Vector3d GetPointVelocity(const Vector3d &p) const = 0;
virtual Vector3d GetPointAcceleration(const Vector3d &p) const = 0;
virtual Matrix3x3 GetR() const = 0;
virtual Vector3d GetPosition() const = 0;
virtual double GetRestitution() const = 0;
virtual double GetFriction() const = 0;
```

The `Collidable` interface needs a way to handle the forces and impulses generated during contact and collision resolution. The methods `ApplyImpulse()` and `AddForce()` are defined for this reason. Additionally, each interface has to define an `ApplyTestImpulse()` method that applies an impulse to a temporary version of the object's state. After a test impulse is applied, the state will need to be reset to test other impulses in isolation. Therefore, a `Collidable` interface also needs to implement a `ResetTestState()` function that reverts to the pre-test state. These methods are necessary for building the mass matrix needed for the Linear Complementarity Problem Solver and are the next set of functions for the interface listed below.

```
virtual void ApplyTestImpulse(const Vector3d &point,
                              const Vector3d &force) = 0;
virtual Vector3d GetTestPointVelocity(const Vector3d &p) = 0;
virtual void ResetTestState() = 0;
virtual void ApplyImpulse(const Vector3d &point,
                          const Vector3d &force) = 0;
virtual void AddForce(const Vector3d &point,
                      const Vector3d &f) = 0;
virtual void AddContactForce(const Vector3d &point,
                             const Vector3d &force) = 0;
```

Finally, the `Collidable` interface ensures that the object defines functions that deal with constraints. For example, if two distinct `Collidable` objects are connected through a constraint system, a force or impulse on one will affect the other. Without this functionality, only direct contact events would be seen. This is crucial for

articulated structures, where each link can be treated as a separate collidable object and the articulated structure as a whole is treated as a constraint. Such functionality can also be used to enforce arbitrary constraints such as springs, pin constraints, or distance constraints. The function `IsRelated()` returns true if the object passed to it is connected to the current `Collidable` object via a constraint. Another method, `GetConstraint()` returns the constraint containing both the current object and the object passed through the parameter. These methods round out the `Collidable` interface.

```
    virtual bool IsRelated(Collidable *co) = 0;
    virtual Constraint* GetConstraint(Collidable *co) = 0;
  };  //end Collidable
```

4. `Constraint`

An object that implements the `Constraint` interface affects the collision behavior of other `Collidable` objects. This interface allows for proper collision response between the links of an articulated structure and objects connected via arbitrary springs, hinges, and pins. The collision detector treats `Constraint` objects separately from `Collidable` objects (this does not mean that a class cannot implement both a `Collidable` and a `Constraint` interface). During specific parts of the simulation, the state of the `Constraint` objects is verified to ensure that no velocity or acceleration constraints are broken. `EvaluateVelocityConstraints()` checks for velocity violations and `EvaluateAccelerationConstraints()` checks for acceleration violations. For articulated bodies, internal constraints such as joint limits are handled in these two function calls. Joints bouncing against their limits would be a velocity violation while joints at rest accelerating through their limits would be an acceleration violation. These constraints potentially affect other contacts (i.e., objects resting on links that are themselves at rest on their joint limits), so care must

be taken to ensure that all the constraints are solved for at once. The class definition for the `Constraint` interface follows.

```
class Constraint {
public:
  virtual int EvaluateVelocityConstraints() = 0;
  virtual int EvaluateAccelerationConstraints() = 0;
  virtual void ApplyVelocityResolution() = 0;
  virtual void ApplyAccelerationResolution() = 0;

  virtual void AddConstraint(Constraint *co) = 0;
  virtual void ResetTestState() = 0;
  virtual void ApplyTestImpulse(const Vector3d &point,
    const Vector3d &force, const int &link) = 0;
  virtual void ApplyTestImpulse(Collidable *a, ICollidable *b,
    const Vector3d &point, const Vector3d &force) = 0;
};
```

When objects connected by constraints are affected by impulses and forces resulting from collision or contact resolution, those impulse and forces potentially have to be distributed across all objects involved in the constraint. For example, an impulse on one link in a chain can affect the velocities of all links in the chain. For that reason, each constraint class must implement an `ApplyVelocityResolution()` function and an `ApplyAccelerationResolution()` function. These functions apply impulses and forces to the constrained objects respectively.

Finally, just as the `Collidable` interface defines functions for applying test impulses and resetting the test state, a `Constraint` object must be able to do the same. Since an impulse on an object may affect other objects under the constraint, these functions ensure that the impulse is applied correctly and all the effects are visible. The functions have the same names as those for the collidable interface: `ApplyTestImpulse()` and `ResetTestState()`.

5. `Renderable` and `SidRenderable`

There are two primary interfaces for displaying an object. If the framework is used by itself as a standalone application, the `Renderable` interface ensures that that object

can be rendered to the framebuffer. Additionally, the framework can be used with an immersive rendering engine. The rendering of objects on an immersive display is handled differently using the `SidRenderable` interface.

The `Renderable` interface is used for the standalone physics engine and defines a `Render()` function and a rendering quality variable. How the object is rendered, however, depends entirely on the class's implementation of the function. This creates an easy way for the physics engine to separate all the objects that need to be drawn. When a frame render is requested, the physics engine simply iterates through the list and calls the `Render()` method of all the renderable objects. The `Renderable` interface is very simple and is shown below.

```
class Renderable {
public:
  virtual void Render() = 0;
protected:
  RenderQuality renderQuality;
};
```

The `SidRenderable` interface is designed to work with the immersive engine and is one of the few classes that deal directly with the immersive side (the 'Sid' prefix stands for Spatially Immersive Display). Instead of defining a rendering function like the `Renderable` interface, `SidRenderable` essentially gives functionality for the immersive engine to load a model separately. This model is then treated like any other model in the immersive engine, except that its transformation information is controlled by the physics engine. The `SidRenderable` interface is shown below.

```
class SidRenderable {
public:
  virtual const char *GetObjFile() const = 0;
  virtual const char *GetShaderName() const = 0;

  virtual void LinkToTransform(\g3d::Empty *empty) = 0;
  virtual void GetShaderProperties(\g3d::Shader *shader) = 0;
};
```

The `SidRenderable` interface defines a set of functions that get the path to an OBJ file used for displaying the model, the shader name, shader attributes,

and lastly, a function that passes in a pointer to a transform node. The function `LinkToTransform()` is used to connect the transformation of the model in the immersive engine to the object in the physics engine. The immersive engine knows nothing about the fact that a model is being manipulated by another system; it merely displays the model using the given position and orientation.

6. NetSync

The NetSync interface is not necessary in the standalone version; however, it becomes critical in the immersive environment. Essentially, this interface allows an object to be synchronized over a network. The interface itself is very simple: any class that implements it must define a function called `GetData()` and `ReadData()`. The first method asks the object to provide synchronization information for transfer over a network. The second method is used to apply information received from a network socket. The header and the methods and are presented in the following class definitions:

```
class NetSyncHeader {
public:
  char name[64];
  size_t blockSize;
};

class NetSync {
public:
  virtual bool GetData(NetSyncHeader &header, char* &data) = 0;
  virtual bool ReadData(const NetSyncHeader &header, char *data) = 0;
};
```

The synchronization follows a very simple scheme as shown in Figure 13. A `NetSyncHeader` class is declared that has a fixed size character name field and a length field. Since the size is of the header is known ahead of time, we can always tell if we receive a partial message. The header will have the name of the object to which the data is addressed and the length of the entire message. If the received packet is smaller than the header size, the full header has not been received; if the packet is

smaller than the length specified in the header, then the entire message has not been received.

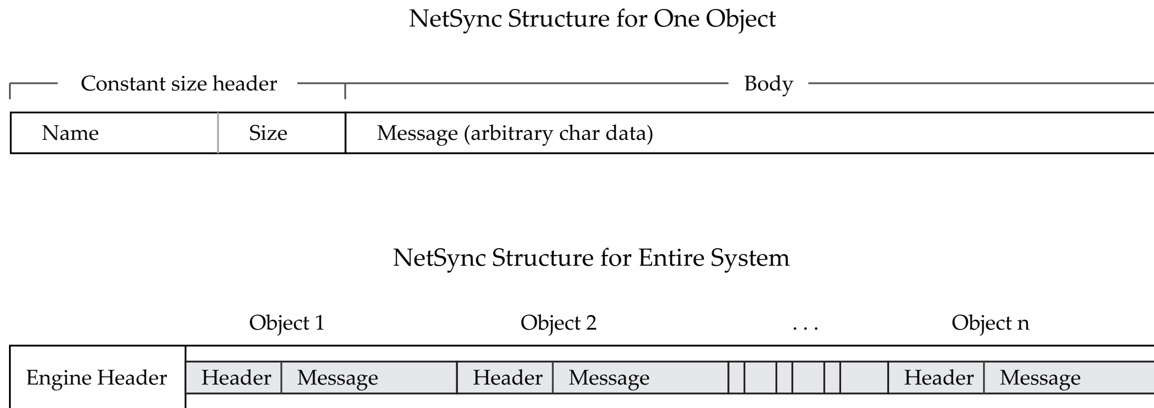

Fig. 13. An example of `NetSync` structure.
The header is of constant size for all messages. The message body for the engine consists of header/message combinations that are processed in sequence.

The message data can be structured in any manner necessary. The only requirement is that when the message is passed to the object specified in the name field, that object knows how to deal with it. The `Engine` class itself implements this interface and constructs the message such that the first header has "Engine" in the name and the length of the entire message in the length field. When the packet is sent across the network, the `Engine` header will be the first one received. The `Engine` object then iterates over all objects that implement the `NetSync` interface and calls their `GetData()` function. Each of those objects returns a header with their name, the length of the individual message, and the message itself. The `Engine` object then collects all of the messages keeping track of the total length, puts them into a single message with the `Engine` header at the front, and returns that.

When the `Engine` object on the client side receives the message, it decodes the message by looking at the individual headers and passing each of the messages to their

respective objects. Using this method, a single physics engine on the server runs the simulation while updating the other client engines that are part of the distributed environment.

B. The Physics Engine

The physics engine is built upon heavy use of the interfaces depicted in Figure 12 and described in detail above. Figure 12 also shows the three primary classes that make up the essential functionality of the dynamic system. These classes are `Engine`, `SimulationSystem`, and `CollisionDetector`. The main `Engine` class is responsible for managing all objects and interfaces and delegates the appropriate objects to the simulation system and the collision detector. Each object inherits from a base `Object` class that ensures that the physics engine can determine what interfaces that object implements.

1. The `Object` Class

The `Object` class serves as a base for most classes that are handled by the physics engine. This class provides some simple common functionality such as getting and setting the object's name, the class name, and object ID. Additionally, the Object class declares a critical function `ImplementsInterface()` [17]. Using this method, an object can specify which interfaces it inherits from and returns a pointer to that interface. This mechanism allows all objects that are added to the physics engine to be dealt with automatically based on the interfaces they implement.

Upon addition of an object, the physics engine goes through the available list of interfaces and checks to see if the object implements a particular interface. If it does, the engine takes the pointer to that interface and delegates it into the appropriate

list. For example, if an object implements `Renderable`, the engine places it in a list for rendered objects. This list is traversed upon drawing a frame. If the object implements the `Collidable` interface or `Integratable` interface, that object is added to the collision detector or the simulation system classes respectively. The rest of the interfaces are treated similarly. Note that since a class can easily inherit from multiple interfaces, an object of that class can wind up in many places. For example, a rigid body object can be integrated, rendered, checked for collisions, and synchronized over a network. The engine only has pointers to the interfaces that the rigid body implements and that is all the information it needs.

2. The Simulation System

The `SimulationSystem` class is responsible for advancing objects' states based on the current state of the system. The simulation system does that by working with objects implementing the `Integratable` interface.

When a new object is added to the physics engine, the engine checks to see if it implements the `Integratable` interface. If so, that object is passed on to the `SimulationSystem`. The object is guaranteed to have the necessary simulation states defined. When the physics engine needs to update the state of the simulation by a timestep, the simulation system integrates the state of all `Integratable` objects. The basic class definition for the `SimulationSystem` class is

```
class SimulationSystem : public Object {
public:
  SimulationSystem();
  ~SimulationSystem();
  void AddNewObject(Integratable *io);
  void RemoveObject(Integratable *io);
  void SystemDynamics();
  void Integrate(const double &delta);
  void Commit();
protected:
  list <Integratable*> integObjects;
  list <Integratable*> liveObjects;
```

```
  int integratorOrder;
};
```

SimulationSystem inherits from Object and is a member of the Engine class. The three important functions in this class are SystemDynamics(), Integrate(), and Commit(). SystemDynamics() loops through all live objects in the simulation and calls their AccumulateForces() function. This updates the state of the simulation to ensure that all objects have the most current forces acting on them before any derivatives are taken. Integrate() uses a method of integration specified by integratorOrder to compute the next state for each object based on the timestep. And finally, Commit() advances the current state to the next state.

The simulation system keeps track of two lists of Integratable objects. The first list, integObjects, holds objects that require integration. This list is parsed during the Integrate() call. The second list, liveObjects, is a superset of integObjects and contains all objects that influence the integration of the system, but do not necessarily have to be integrated. For example, a "springy mesh" object would be a live object, as its AccumulateForces() function would provide particles with the spring forces. The object itself, however, does not have any state information and does not need to be integrated.

The integratorOrder property controls what method Integrate() uses. A value of 1 performs an Euler integration and is the default setting. The code for an Euler integration is as follows:

```
list <Integratable*>::iterator i;
SystemDynamics();
for(i = integObjects.begin(); i != integObjects.end(); i++){
  *(*i)->tempState = *(*i)->nextState;
  *(*i)->tempState.Derivative();
  *(*i)->tempState *= h;
  *(*i)->nextState = *(*i)->currentState;
  *(*i)->nextState += *(*i)->tempState;
  (*i)->UpdateState(STATE_NEXT);
}
```

```
for(i = integObjects.begin(); i != integObjects.end(); i++){
  (*i)->Stabilization(h);
}
```

As noted before, the `SystemDynamics()` function is called before any derivatives are taken. Then, the system iterates through the `integObjects` and performs an Euler step on each element. The use of standard template lists gives us easy management, but requires an extra level of indirection. After the next state is computed, the `UpdateState()` function is called to acquire all the auxiliary properties such as velocities from momenta for rigid bodies and spatial velocities and accelerations for rigid bodies. This is necessary for collision detection and resolution since objects are tested for intersection using the next state positions, velocities, and accelerations. Once all the objects are integrated, the `Stabilization()` function is called to perform any post-integration methods.

During the simulation step, the physics engine will first call `Integrate()` and then `Commit()` to advance the simulation. The simulation step will be discussed in more detail in Section 4 of this chapter.

3. The Collision Detector

To add believability to a physics simulation, the simulation must detect and respond appropriately to collisions. Our physics engine incorporates this functionality into a `CollisionDetector` class. This class deals with the `Collidable` interface and provides a number of intersection methods. The class definition is presented below:

```
class CollisionDetector : public Object, public Renderable {
public:
  CollisionDetector();
  ~CollisionDetector();
  void AddObject(Collidable *obj);
  void AddObject(Constraint *obj);
  void RemoveObject(Collidable *obj);
  void RemoveObject(Constraint *obj);
```

```
    void UpdateConstraintSet(bool &needBacktrack, bool &needImpact);
    void ResolveImpacts(const double &h);
    void ResolveContacts(const double &h);
protected:
    list<Collidable*> activeList;
    list<Collidable*> passiveList;
    list<Constraint*> constraintObjectList;
    vector<Contact> contacts;
    vector<Contact> restingContacts;

    Wm4::LCPSolver *lcpSolver;
    double collisionThreshold;
};
```

The `CollisionDetector` class inherits from the `Object` class and implements the `Renderable` interface. We found it very useful to visualize contact information as bodies collide. The class simply defines a `Render()` function that renders contact points, normals, velocities, and forces when the simulation is running in debugging mode.

Objects implementing the `Collidable` interface are stored in `activeList` or `passiveList`, depending on whether collision is enabled for that object. The variable `constraintObjectList` holds objects implementing the `Constraint` interface. The total set of constraints is represented as a vector of contacts in the collision detector. Initially, all contacts are treated as colliding contacts. After collision resolution, contacts are tested for acceleration violations, and are moved to the `restingContacts` array as appropriate. The `LCPSolver` class is used to compute the solution to the Linear Complementarity Problem as described by Eberly [18]. Finally, a variable that determines the collision tolerance is defined. This value is used when determining if two objects collide or intersect, as well as in other calculations.

The first four methods in the `CollisionDetector` class, aside from the constructor and destructor, add and remove objects that implement `Collidable` and `Constraint` interfaces to the collision detector. The next three methods are used in the simulation step.

a. Collision Detection

`UpdateConstraintSet()` performs several tasks. The current implementation performs an exhaustive set of intersection tests. The tests check to see which of the collidable objects intersect by running the `Intersect()` method for each pair. Class property `collisionThreshold` is used to determine whether two objects intersect. The two parameters to the function, `needBacktrack` and `needImpact`, are set based on the return values of the intersect tests. If any of the tests return a value of +1 for intersection, the `needBacktrack` boolean is set to `true`. If contacts are present, i.e. the return value for at least one test is 0, then the `needImpact` is set to `true`. The booleans tell the physics engine if the integration has proceeded too far and intersections are occurring, or if contacts are present that have to be resolved.

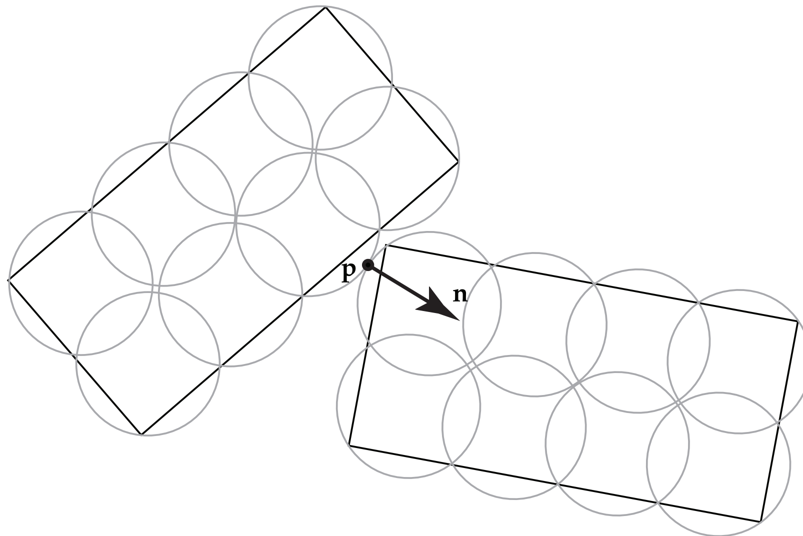

Fig. 14. Two sphere-trees in contact.
We get the contact point **p** and contact normal **n** from the sphere-tree intersection.

Since we are mainly dealing with polyhedral objects, we need a method of determining if two bodies collide. For this thesis, we have decided to use sphere-tree

bounding volumes for our collision detection method. As can be seen in Figure 14, we get the collision information directly from the spheres instead of actually testing intersecting geometry. There are several reasons for using sphere trees. First, the sphere is a very simple primitive. Second, sphere-sphere intersection tests are very fast and always yield some sort of collision information. Even if two spheres are deeply intersecting, we can still get some approximate contact point and normal information (Figure 15). Third, intersecting sphere tree hierarchies allow for quickly culling away large portions of the model that cannot be intersecting. And finally, this method allows us to represent any closed model without being limited to any particular shape or convex geometry.

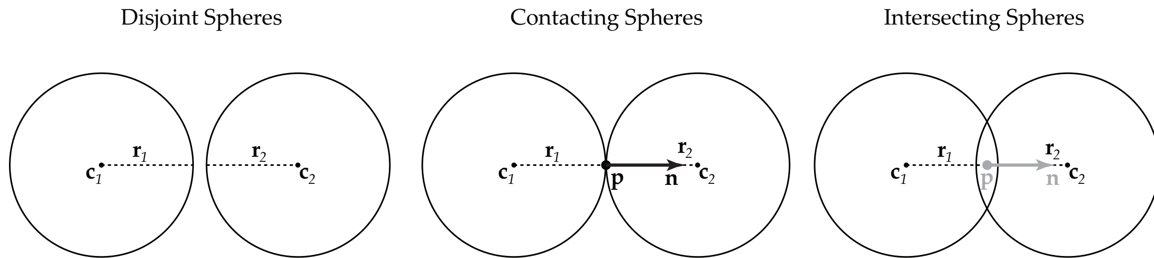

Fig. 15. Sphere contact and intersection.
If the distance between the radii of two spheres is smaller than some small value ϵ, the spheres are in contact. Note that even if spheres are intersecting, we can still get a good guess at the contact point and normal.

We use Bradshaw's adaptive medial axis method for sphere-tree construction [19, 20]. This method initially constructs a medial axis for a closed polyhedral body based on surface point samples. Initial guess spheres are fitted to the vertices of the medial axis and are refined using various methods. Figure 16 demonstrates that this method provides fairly tight-fitting trees around the geometry. Depending on the depth of the tree, the contact information is a good approximation to the real surface.

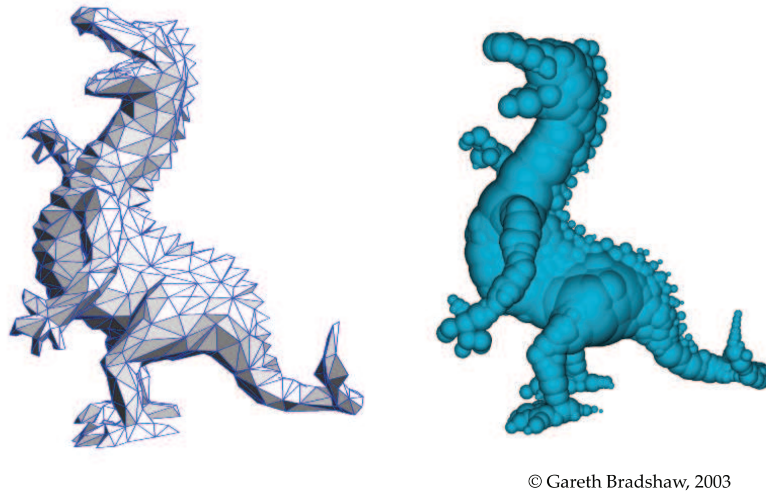

Fig. 16. Third sphere-tree level of a dragon model.

There are, of course, some drawbacks to using spheres for collision detection. Accurately representing objects with large planes is difficult. Collisions on sides of large boxes give slightly random bounce directions due to the underlying sphere representation. However, if the objects do not have large planar sides, the sphere-tree representation is quite effective. The other major drawback of this method that unlike a polygonal mesh, the sphere is a continuous surface. Figure 17 illustrates one problem that this creates. During resting contact resolution, contact forces may not be enough to keep a sphere from interpenetrating. If a sphere is spinning in place with no friction, the contact point will have centripetal acceleration $\mathbf{a}_c$ upwards, offsetting acceleration due to gravity $\mathbf{a}_g$. The contact force is then just large enough to keep that point from interpenetrating. However, the contact point is rotating about the body's center of mass while the center falls through the plane. Resolving continuous surface contacts is a complex topic which increases in complexity when multiple continuous objects are involved [21, 22].

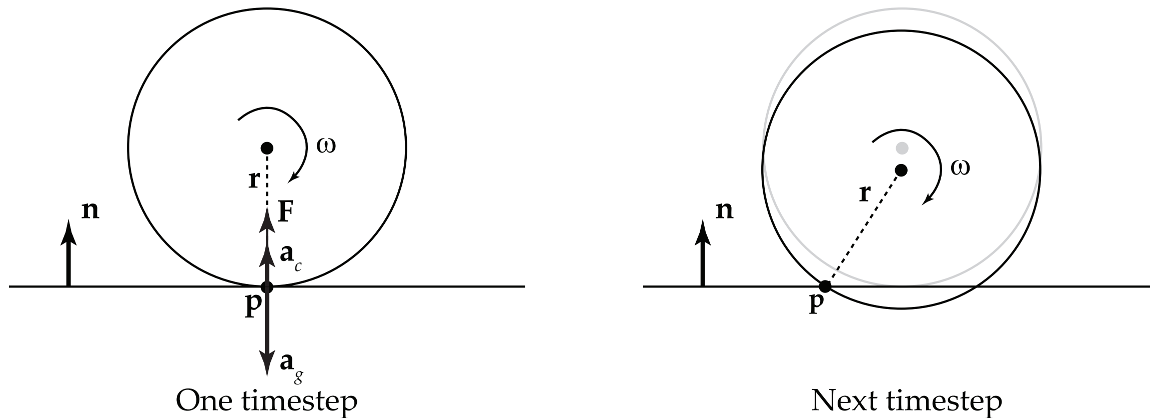

Fig. 17. Spinning sphere problem.
The contact point, **p**, is partially accelerating upward due to centripetal forces. The contact force **F** is enough to keep **p** from interpenetrating on the next step, but not the rest of the sphere.

b. Colliding Contact Resolution

The next function, `ResolveImpacts()`, is called if `needBacktrack` is `false` and `needImpact` is `true`. During intersect testing, contacts are collected into the `contacts` array. The set of contacts is then processed for collision resolution and the solution is found at once using the Linear Complimentary Problem solver. However, we generalize the solution to the `Collidable` interface so that any entity that properly defines that interface can be used.

The system of equations for the Linear Complementarity Problem has the following form:

$$\mathbf{Kf} + \mathbf{v}_{rel}^{-} = \mathbf{v}_{rel}^{+},$$

where $\mathbf{K}$ is a matrix of $n \times n$ size, $\mathbf{f}$ is a vector of impulses, $\mathbf{v}_{rel}^{-}$ is a vector of pre-impulse velocities, and $\mathbf{v}_{rel}^{+}$ is a vector of post-impulse velocities. The solver will find solutions for $\mathbf{f}$ and $\mathbf{v}_{rel}^{+}$ that satisfy the LCP constraints. For rigid bodies, computing

the matrix $\mathbf{K}$ involves calculating mass and inertial properties. For other dynamic objects, such as articulated bodies, this becomes more difficult, and another method has to be developed. In order to build the matrix, we use the `ApplyTestImpulse()` methods of the `Collidable` interface. Each matrix component k_{ij} gives the effect of a unit impulse in the normal direction of contact i on the relative velocity of contact j. The pseudocode for building the matrix is presented in Algorithm 2.

Algorithm 2 Pseudocode for BuildMatrix.

procedure BUILDMATRIX
- n = number of contacts
- **for** $i = 1$ to n **do**
  - A_i.ApplyTestImpulse($\mathbf{p}_i$, $\mathbf{n}_i$)
  - B_i.ApplyTestImpulse($\mathbf{p}_i$, $-\mathbf{n}_i$)
  - **for** $j = i$ to n **do**
    - $\mathbf{v}_p^A = A_j$.GetTestPointVelocity($\mathbf{p}_j$)
    - $\mathbf{v}_p^B = B_j$.GetTestPointVelocity($\mathbf{p}_j$)
    - $v_{rel} = (\mathbf{v}_p^B - \mathbf{v}_p^A) \cdot \mathbf{n}_j$
    - $k_{ij} = k_{ji} = v_{rel}^t - v_{rel_j}$
  - A_i.ResetTestState()
  - B_i.ResetTestState()

The first loop iterates through the contacts and applies an equal but opposite test impulse to each of the bodies in the contact. The subscript refers to a contact, so A_i, $\mathbf{p}_i$, and $\mathbf{n}_i$ would be body A, point, and normal of contact i. Once the test impulse along the contact normal is applied, the velocity of the two bodies changes. The inner loop iterates through the contacts to check how the relative velocity was affected in all the other contacts. Here, $\mathbf{v}_p$ is the point velocity of body A or B as denoted by the superscript, v_{rel_j} is the relative velocity of the original contact, and v_{rel} is the relative velocity resulting from the test impulse. Note that if neither of the bodies in contact i are related to bodies in contact j, the test impulse has no effect and k_{ij} is zero. Moreover, the matrix is diagonally symmetrical, so we only need to

compute half of its components.

Aside from the matrix, we need the $\mathbf{v}_{rel}^{-}$ vector, the relative pre-impulse velocities of each of the contacts. $\mathbf{v}_{rel_i}^{-} = 0$ if $v_{rel_i} > 0$ and $\mathbf{v}_{rel_i}^{-} = \epsilon v_{rel_i}$ if $v_{rel_i} < 0$. The Linear Complementarity Problem solver gives us $\mathbf{f}$, which contains f_1 through f_n. Then, the impulse for each body in a contact is

$$\mathbf{f}_i^A = f_i \mathbf{n}_i \,,$$
$$\mathbf{f}_i^B = -f_i \mathbf{n}_i \,.$$

The `ApplyImpulse()` function of the Collidable interface is used to apply the impulse at the contact point for each body.

c. Resting Contact Resolution

The last critical function of the `CollisionDetector` class is `ResolveContacts()` and differs from `ResolveImpacts()` in that it deals with resting contacts instead of colliding contacts. Once again, we have a Linear Complementarity Problem, this time described by

$$\mathbf{Kf} + \mathbf{a}_{rel}^{-} = \mathbf{a}_{rel}^{+},$$

which determines the forces needed for resting contact resolution. As mentioned in section III.A.3 and section III.B.6, the relative velocity of each contact point must be zero within some tolerance. If the contact list is unchanged from the impact resolution step, the same matrix $\mathbf{K}$ can be used. Otherwise, $\mathbf{K}$ will be a subset of the other matrix since some contacts may be separating. We still need to determine $\mathbf{a}_{rel}^{-}$. $\mathbf{a}_{rel_i}^{-} = 0$ if $a_{rel_i} > 0$ and $\mathbf{a}_{rel_i}^{-} = a_{rel_i}$ otherwise.

Once we have the matrix and the vector of initial accelerations, we can use the Linear Complementarity Problem solver to get the contact forces and the resulting accelerations. The function `AddForce()` is used to apply those forces to each body

at the points.

d. Friction

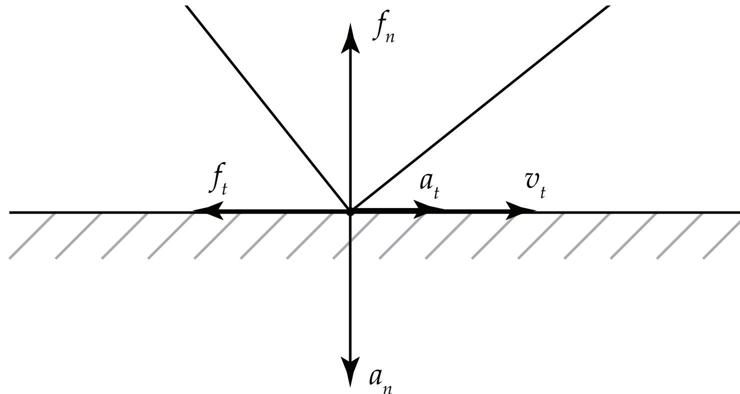

Fig. 18. Contact with friction.

Friction is an important physical behavior necessary for visually realistic simulation. Figure 18 shows some common vectors used during friction contact. To accommodate friction, we extend the matrix **K** built up in the collision response stage. We do this to introduce extra frictional constraints. Kokkevis formulates the appropriate constraints in the following way [5]:

$$\begin{aligned} a_n - a_n^d \geq 0 &\quad \text{complementary to} \quad f_n \geq 0 \\ (a_t - a_t^d) + \lambda \geq 0 &\quad \text{complementary to} \quad f_t \geq 0 \\ \mu f_n - f_t \geq 0 &\quad \text{complementary to} \quad \lambda \geq 0 \end{aligned}$$

The variables a_n, a_t, and a_n^d, a_t^d are the actual and desired accelerations along the normal and tangent directions. f_n and f_t are the force magnitudes along those directions. μ is the coefficient of friction and λ is a Lagrange multiplier that is used to limit the magnitude of tangential friction. We know actual accelerations

and we can calculate the desired accelerations. $a_n^d = 0$ since we want to prevent objects from accelerating toward each other. Dynamic friction will try to cancel out all of the tangential velocity in one timestep h, therefore, $a_t^d = -v_t^-/h$. For static friction, $a_t^d = 0$ as friction will try to prevent the point from accelerating. The third constraint will limit tangential forces to the magnitude of the normal force scaled by the coefficient of friction.

Having three constraints for each contact implies that there will be three equations for each contact in the Linear Complementarity Problem. A single contact in matrix form is formulated as

$$\begin{bmatrix} k_{nn} & k_{tn} & 0 \\ k_{nt} & k_{tt} & 1 \\ \mu & -1 & 0 \end{bmatrix} \begin{bmatrix} f_n \\ f_t \\ \lambda \end{bmatrix} + \begin{bmatrix} a_n - a_n^d \\ a_t - a_t^d \\ 0 \end{bmatrix} = \begin{bmatrix} a_n^+ \\ a_t^+ \\ \lambda^+ \end{bmatrix}.$$

We now have some new variables: k_{nn}, k_{tn}, k_{nt}, k_{tt}. The first, k_{nn}, refers to the response of relative normal velocity to a unit test impulse along the normal direction. Likewise, k_{tt} is the response of relative tangential velocity to a test impulse along the tangential direction. k_{nt} and k_{tn} are the responses of relative normal velocity to tangential test impulses and vice-versa. It is these new variables that we must compute to build our new friction matrix $\mathbf{K}^F$.

We have structured $\mathbf{K}^F$ mirroring the 3×3 matrix. The entire matrix has dimensions $3m \times 3m$ where m is the number of contacts. It consists of nine submatricies:

$$\mathbf{K}^F = \begin{bmatrix} \begin{bmatrix} k_{n_1n_1} & \cdots & k_{n_mn_1} \\ \vdots & \ddots & \vdots \\ k_{n_1n_m} & \cdots & k_{n_mn_m} \end{bmatrix} & \begin{bmatrix} k_{t_1n_1} & \cdots & k_{t_mn_1} \\ \vdots & \ddots & \vdots \\ k_{t_1n_m} & \cdots & k_{t_mn_m} \end{bmatrix} & \mathbf{0} \\ \begin{bmatrix} k_{n_1t_1} & \cdots & k_{n_mt_1} \\ \vdots & \ddots & \vdots \\ k_{n_1t_m} & \cdots & k_{n_mt_m} \end{bmatrix} & \begin{bmatrix} k_{t_1t_1} & \cdots & k_{t_mt_1} \\ \vdots & \ddots & \vdots \\ k_{t_1t_m} & \cdots & k_{t_mt_m} \end{bmatrix} & \mathbf{1} \\ \begin{bmatrix} \mu_1 & & \\ & \ddots & \\ & & \mu_m \end{bmatrix} & -\mathbf{1} & \mathbf{0} \end{bmatrix}$$

Each submatrix has dimensions $m \times m$, including the zero matrix $\mathbf{0}$ and the identity matrix $\mathbf{1}$. Each k has a subscript that denotes what contact velocity and what test impulse is being compared. For example, $k_{n_1t_3}$ would show the response of contact 3 tangential relative velocity to an impulse along contact 1 normal. The lower left matrix is an identity matrix with the coefficients of friction along the diagonal.

Again, this matrix can be used for both colliding contacts and resting contacts. For colliding contacts,

$$\mathbf{v}_{rel}^- = \begin{bmatrix} v_{n_1}^-, & \cdots, & v_{n_m}^-, & v_{t_1}^-, & \cdots, & v_{t_m}^-, & \lambda_1, & \cdots, & \lambda_m \end{bmatrix}^T,$$

where v_{n_1} through v_{n_m} are defined as described earlier. v_{t_1} through v_{t_m} are the pre-impulse tangential velocities. For resting contact,

$$\mathbf{a}_{rel}^- = \begin{bmatrix} a_{n_1}^-, & \cdots, & a_{n_m}^-, & a_{t_1} - a_{t_1}^d, & \cdots, & a_{t_m}^- - a_{t_m}^d, & \lambda_1, & \cdots, & \lambda_m \end{bmatrix}^T,$$

where $a_{t_1}^d$ through $a_{t_m}^d$ are defined as described at the beginning of this chapter. The pseudocode for building the friction matrix is presented in Algorithm 3.

Algorithm 3 Pseudocode for BuildMatrixFriction.

```
procedure BuildMatrixFriction
    n = number of contacts
    for i = 1 to n do
        A_i.ApplyTestImpulse(p_i, n_i)
        B_i.ApplyTestImpulse(p_i, −n_i)
        for j = i to n do
            v_p^A = A_j.GetTestPointVelocity(p_j)
            v_p^B = B_j.GetTestPointVelocity(p_j)
            v_rel^n = (v_p^B − v_p^A) · n_j
            v_rel^t = (v_p^B − v_p^A) · t_j
            k_{n_i n_j} = k_{n_j n_i} = v_rel^n − v_{rel_j}^n
            k_{n_i t_j} = k_{t_j n_i} = v_rel^t − v_{rel_j}^t
        A_i.ResetTestState()
        B_i.ResetTestState()
        A_i.ApplyTestImpulse(p_i, t_i)
        B_i.ApplyTestImpulse(p_i, −t_i)
        for j = i to n do
            v_p^A = A_j.GetTestPointVelocity(p_j)
            v_p^B = B_j.GetTestPointVelocity(p_j)
            v_rel^n = (v_p^B − v_p^A) · n_j
            v_rel^t = (v_p^B − v_p^A) · t_j
            k_{t_i n_j} = k_{n_j t_i} = v_rel^n − v_{rel_j}^n
            k_{t_i t_j} = k_{t_j t_i} = v_rel^t − v_{rel_j}^t
        A_i.ResetTestState()
        B_i.ResetTestState()
        fill in μ_i and identity matricies
```

Even though the code is longer, the major difference from the frictionless scenario is that another test impulse is needed along the tangential direction. Additionally, an impulse along one direction now produces two coefficients for each contact. Here, $v^n_{rel_j}$ and $v^t_{rel_j}$ are the original relative velocities of contact j along the normal and tangential directions. v^n_{rel} and v^t_{rel} are the relative velocities resulting from the test impulse. The rest of the matrix is easy to fill. Once the matrix and the vectors are built, the solver can give us the impulses and forces needed to resolve collisions and contacts with friction.

4. The Simulation Step

All of the elements described in this section are finally brought together in the physics engine under the update loop. The update loop takes a simulation step and performs several tasks. First, it integrates the system based on the requested timestep. Then, it checks for contacts or intersections. If intersections occur, the system backtracks until only contacts remain. Once only contacts are present, the collision detector resolves the impacts and resting contacts, and the simulation continues. The pseudocode in Algorithm 4 is based on Kokkevis's model for articulated dynamics [5] and has been adapted for a general system.

Whenever the simulation loop starts, we always know the current state of the entire system. The call to `Integrate()` invokes the system dynamics function to get all the appropriate accelerations and computes the next state for each object. At that point, we know all the positions, orientations, velocities, and accelerations. If any contacts are found, the collision detector deals with them using those attributes. A successful `Commit()` call advances the current state of each object to the next state.

Algorithm 4 Pseudocode for the Simulation Step.

```
procedure SimulationStep(h)
    while h > 0 do
        hTry = h ; repeat = true
        while repeat do
            simSystem.Integrate(hTry)
            needBacktrack = false ; needImpact = false
            collisionDetector.UpdateConstraintSet(needBacktrack, needImpact)
            if needBacktrack then
                hTry = hTry/2
                continue
            if needImpact then
                collisionDetector.ResolveImpacts()
                collisionDetector.ResolveContacts()
            h = h − hTry
            simSystem.Commit()
            repeat = false
```

C. guppy3d and Dynamics

It was possible to incorporate the standalone physics code into the immersive environment with little trouble. Since the dynamics run independently, it was simply a matter of providing the correct interface between the immersive engine and the physics engine. The primary interaction consists of `SidRenderable` methods that provide guppy3d the location of the physics models and their shader properties.

Essentially, the dynamics work in the following manner. The immersive engine is initialized with the standard scene file. This file includes lights, camera, and user information. Additionally, it has any non-dynamic geometries. Then, the immersive engine initializes an instance of the physics engine, and loads in a dynamic scene file. This file defines rigid bodies and articulated bodies, as well as any static geometry in the world. Then, using the `SidRenderable` interface, guppy3d gets locations of the dynamic models and their shader properties.

With this method, the physics engine is not overly intrusive in GUPPY3D, as the engines only interact in a few places. GUPPY3D has to initialize the physics engine, tell it which scene file to load data from, and call its `Update()` method. GUPPY3D has been modified to run a dynamic scene using the following steps.

1. Any previously running GUPPY3D processes are terminated across the server and all clients.

2. Each machine performs the following steps:

   (a) GUPPY3D application is launched with the GUPPY3D scene and the dynamics scene files as parameters.

   (b) The GUPPY3D scene file is parsed for geometry, lights, camera, and user information. This geometry will make up the non-dynamic elements of the scene.

   (c) An instance of the physics `Engine` class is created.

   (d) The physics scene file is parsed. Any light or camera information is ignored. Dynamic objects in the physics file are added to the engine.

   (e) All objects implementing `SidRenderable` interface in the physics engine are added to the GUPPY3D scene.

3. The server physics engine is designated as the main engine; the rest are marked as clients.

4. Server begins its display loop.

During each iteration of the display loop, the server calls the `Update()` function of its physics engine with the appropriate timestep. The physics engine performs a simulation step, updating the positions and orientations of its dynamic objects. Those

dynamic objects implementing the `SidRenderable` interface update the GUPPY3D transformation properties. The server then proceeds to display the scene.

Additionally, the server acquires a `NetSync` data structure from the physics engine and all of its synchronizable objects. Using the network, the server updates the clients with this information. The clients return their current frame number back to ensure that no frames were dropped. The cycle then repeats.

The networking in GUPPY3D had to be modified to a more generalized scheme. Previously, the immersive engine was only capable of sending camera and user information across the network. It had hard-coded methods for generating and receiving this message. However, the physics engine necessitates a more versatile method of transferring not only camera and user information, but all the dynamic transformations as well. The `NetSync` interface allows us to do just that. In order to incorporate GUPPY3D native camera information into this scheme, we had to create a new object for the physics engine. This object, `SidData`, implements the `NetSync` interface and contains pointers to GUPPY3D camera transformation. After instantiating the `Engine` class, a `SidData` object is initialized and added to the physics engine. The physics engine's `NetSync` message now contains camera and dynamic information. After the clients receive the physics engine message, they read the `SidData` object to get updated camera data.

CHAPTER V

RESULTS AND DISCUSSION

We have set out to accomplish several goals in this thesis. First, we wanted to supplement existing literature with implementation details. Second, we wanted to design and validate a physics framework built using a well-structured object-oriented approach. The framework should be extensible, allowing users to quickly add new dynamic behaviors. Third, we wanted this framework to function as a standalone application. And finally, we wanted to incorporate the framework into an immersive rendering engine to extend its functionality.

A. Documentation of Concepts and Algorithms

While developing the framework for the physics engine, we have studied the literature on rigid body dynamics and articulated dynamics, as well as hybrid simulation and collision detection and response. We presented a thorough explanation of the basics for rigid body dynamics and articulated dynamics in chapter III.

We define rigid body and articulated dynamics properties necessary for simulation. We explain the basics along with necessary notation such as spatial algebra, give derivations for these properties, and provide pseudocode algorithms for more complex concepts. Additionally, we discuss how both types of dynamics respond to collisions and define the properties needed for applying impulses and forces.

Articulated body collision response in particular is a difficult topic. We explain how to propagate impulses and apply forces to an articulated structure using simple forces. These methods are used to bring rigid body and articulated dynamics collision response into a single system and to detail the actual structure of our framework. We present an extensible way to incorporate other types of dynamic behavior into one

framework through the use of interfaces. Other aspects of the physics engine, such as integration, rendering, and networking are also discussed along with the main `Engine`, `CollisionDetector`, and `SimulationSystem` classes.

We explain how to set up matrices and vectors needed for general contact resolution, in both colliding and resting cases. Friction is incorporated into this model with an extended algorithm for building the friction matrix. We also discuss our collision detection scheme and how we arrive at the contact information. We bring everything together by presenting our pseudocode for a general timestep.

Finally, we discuss how the physics engine ties in with GUPPY3D. The necessary steps for launching an immersive scene with dynamics are listed along with the changes we needed to make to GUPPY3D in order to incorporate our physics engine.

B. The Framework

Our implementation of a generalized physics framework utilizes dynamics concepts discussed in chapter III. The physics engine is built using the interfaces and classes detailed chapter IV. Rigid bodies and articulated dynamics are both supported.

In the following subsections, we present the results from the standalone application utilizing the physics engine. While we were concerned with running speeds of the simulation, our first priority was functionality. Therefore, the simulation could benefit from a number of speedups that will be mentioned as various parts of the framework are discussed.

1. Rigid Bodies

We have been able to simulate the behavior of various rigid bodies. Figure 19 shows one such example. Using sphere-trees for collision detection, we can make any closed

surface into a rigid body. Structuring our code according to the interfaces presented in chapter IV, the state of our rigid bodies is correctly integrated each timestep.

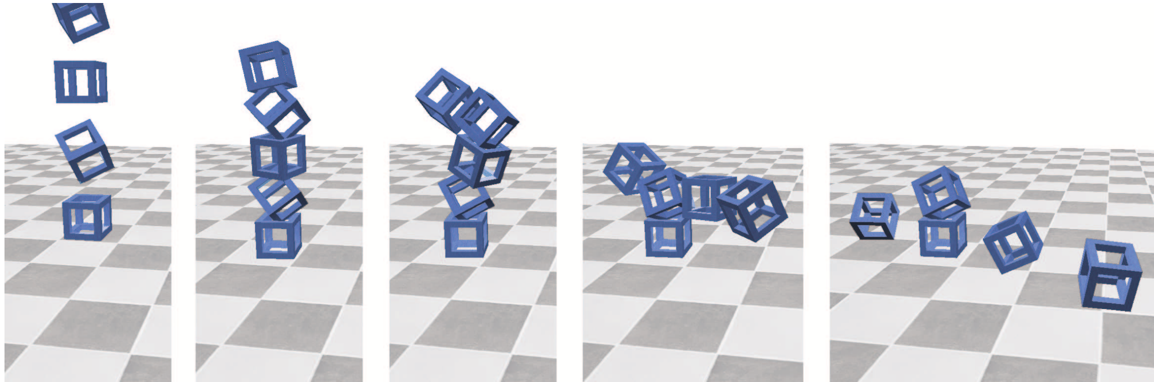

Fig. 19. Rigid bodies in the standalone application.

The collision detector determines the contacts between each pair of bodies and applies the appropriate impulses and forces. The Linear Complementarity Problem solver deals with multiple simultaneous contacts and computes the necessary responses. Our physics engine handles resting contact and multiple stacking bodies. Figure 20 shows one example of stacking blocks. We were also able to simulate a number of other scenarios such as billiards and bowling shown in Figures 21 and 22.

2. Articulated Dynamics

Using the Articulated Body Method, we were able to simulate a number of dynamic scenarios (Figure 23). The structure as shown in Figure 12 on page 39 allows the articulated structures represented by the `KSkeleton` class to move realistically due to gravity and allows the links to collide with ground planes, static geometry, rigid bodies, and other articulated bodies. We have constructed articulated structures such as pendulums, pistons, and simple skeletons.

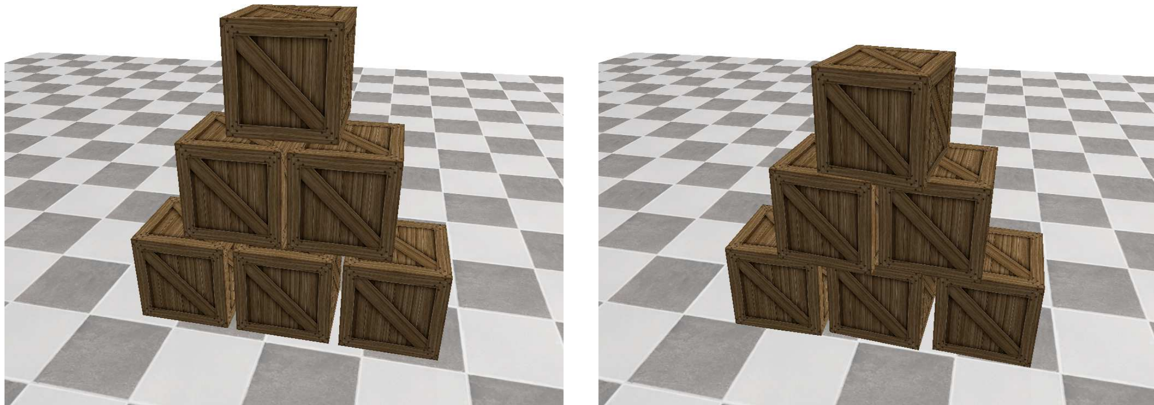

Fig. 20. Rigid bodies in resting contact.

Simulating an articulated structure is a much slower operation than simulating a rigid body. Although the complexity of the Articulated Body Method is $O(n)$, the constant terms slow down the simulation significantly. Spatial matrices and vectors have six dimensions and multiplying 6×6 matrices is costly. There are possible speedups that could be implemented for transformation matrix multiplication [5].

For collision detection, we treat each link as a separate piece of geometry, essentially a rigid body represented by a sphere-tree. The joints collide with static and rigid body geometry and come to rest appropriately. Collision response for articulated dynamics, however, becomes a very expensive operation, since the entire Articulated Body Method has to be called for each test impulse. A faster version of the ABM is used for the test impulses with significantly smaller constant terms. Since many properties have already been calculated in the full version of the ABM, these can be reused for testing impulses. Additionally, because only the effect of test forces on accelerations is desired, the effects of other forces may be ignored. Even with a faster method, multiple simultaneous contacts require numerous test impulses and pose a significant impact to the framerates.

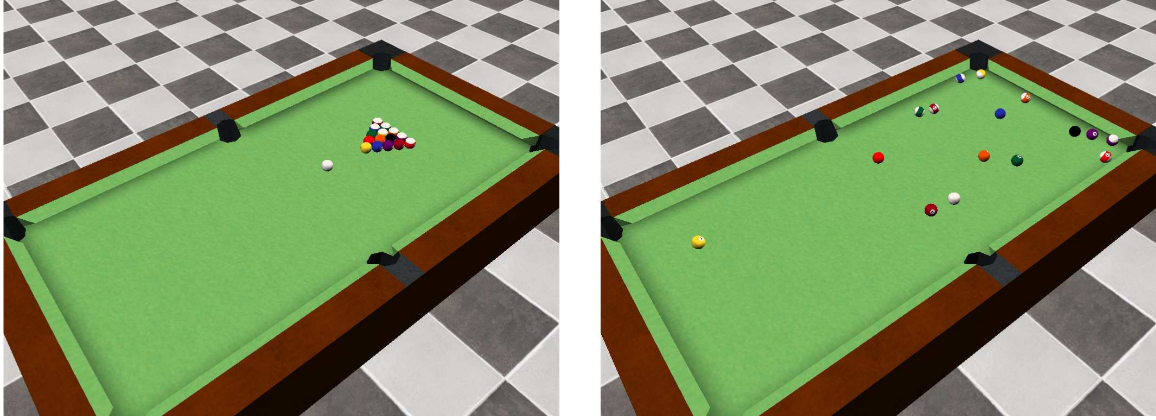

Fig. 21. Rigid bodies in a billiards simulation.

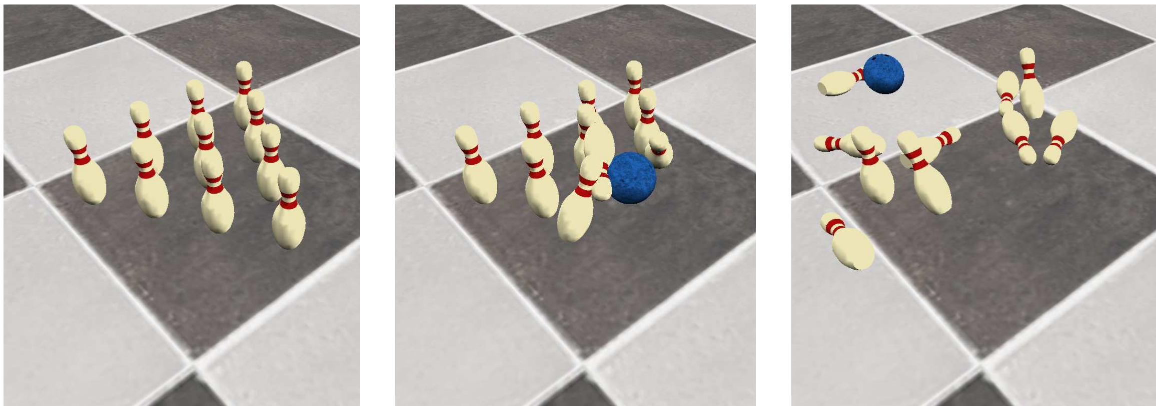

Fig. 22. Bowling.

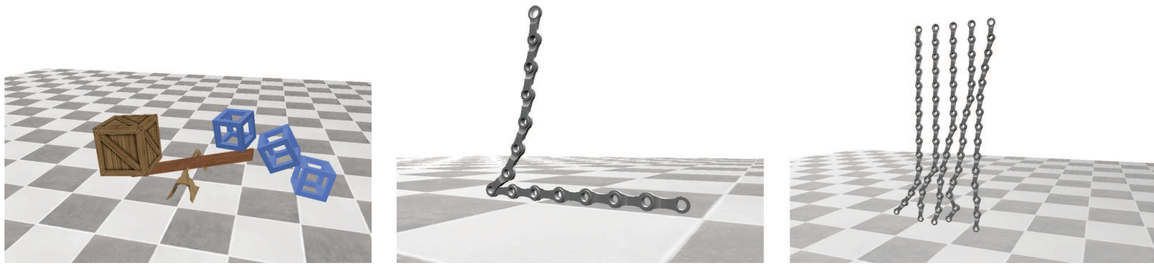

Fig. 23. Various articulated scenarios.

3. Collision Detection and Response

The collision detection method is fairly robust. The sphere-sphere intersection test is fast and provides useful contact information. We had to make one minor change to the sphere-tree construction which can cause a potential slowdown. Figure 24 shows a problem with sphere-trees that do not require a parent sphere to completely encompass all of its children. It is possible for two sphere-tree leaves to be in contact; however, that contact may not be recorded if the parent spheres are not touching or intersecting. Once the parent spheres do touch or intersect, the children have already penetrated beyond the collision threshold. Requiring that parents cover the space of the children is not necessary if the underlying geometry or polygons are tested for intersection; however, our sphere-sphere test requires it.

Although using sphere-trees for collision detection has many advantages, it also presents some adverse behaviors. As we mentioned in the section IV.B.3.b, the sphere itself is a continuous surface. Using large spheres, therefore, is problematic since most of the rigid body ideas are built around polyhedral representations. We also observed a type of "rocking" behavior with objects coming to rest. This would happen if two of the resting contacts happen to be large spheres. The object would roll on those contacts and rock back and fourth. This behavior, however, was only seen with coarse

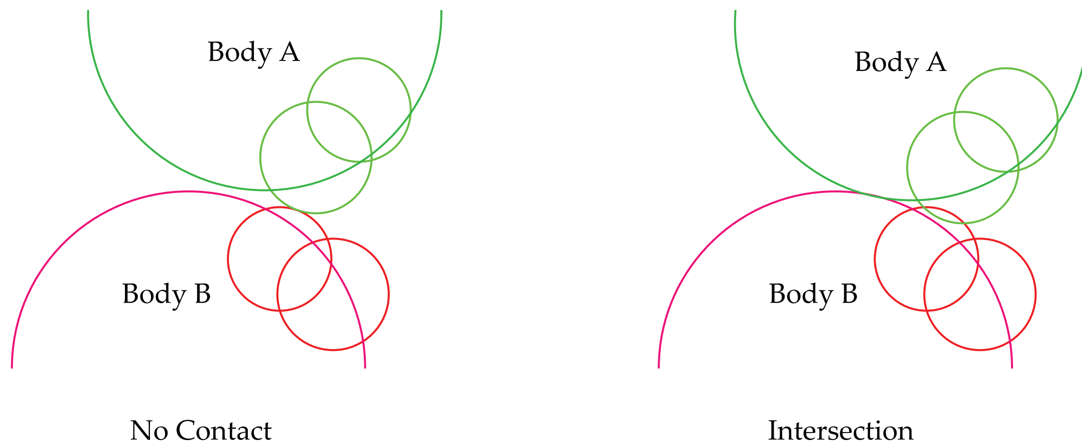

Fig. 24. Parent spheres do not cover all children spheres.
The smaller spheres are children of the larger ones. On the left, the trees are actually in contact, but because the parent spheres do not touch, this contact is not recorded. On the right, the parent spheres are in contact, but the children have already intersected.

sphere-tree representations.

As section IV.B.3.a mentions, we are currently using a brute-force intersection check between all models. This is a very slow procedure and could easily benefit from a coherence-based sweeping method [4]. Such a method would achieve $O(n)$ body-body tests instead of the brute-force $O(n^2)$.

Our collision response involves using a Linear Complementarity Problem solver to determine the impulses and forces necessary for contact resolution. We found this method had advantages and disadvantages. First, it guarantees that impulses and forces will prevent penetrating velocities and accelerations. Setting up the system of equations was fairly easy and using test impulses generalizes the collision response system nicely. Additionally, the code for a Linear Complementarity Problem solver is available on the web.

However, as the number of contacts increases, the speed of finding a solution becomes an issue. Using temporal subdivision further exacerbates the problem as

the solver may need to be called multiple times to resolve multiple substeps. We had tested a method that collects all contact information and attempts to solve everything at the end of the timestep, but that resulted in a slower framerate. The reason was that the solver has one larger system as opposed to several smaller ones.

Friction, therefore, becomes very costly in this scenario. Since each contact is represented by three equations, the size of the system triples. We noticed that it was possible to have hundreds of contacts when several rigid bodies were closely packed together. With friction, the running time of the simulation dropped below interactive speeds. However, for small numbers of bodies, this is not a problem. As Figure 25 shows, friction functions as expected in the simulation. In this case, the block has an initial velocity down the ramp and a non-zero restitution. It slides to a stop as expected.

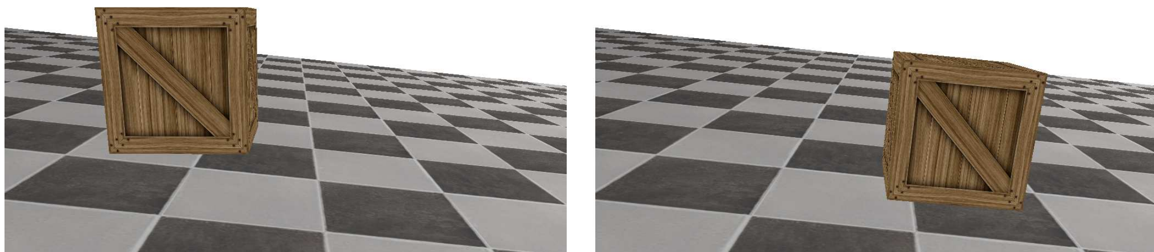

Fig. 25. Block on an inclined plane.

4. Integration

We currently use a basic Euler integration scheme in the physics engine. This has some advantages and some disadvantages.

Euler integration was very easy to implement. Contact point velocities cannot change sign during an integration step. In other words, if two objects have separating

velocities, acceleration will only affect them by the next time step. This helps for resting contact since separating velocity is guaranteed to keep a point separating. Euler integration also guarantees a linear relationship between applied forces and velocities.

However, Euler integration is also inherently unstable for undamped oscillatory systems. This is particularly troublesome for resting contact and swinging motions of articulated dynamics. Objects coming to rest tend to vibrate before settling. Since articulated chains frequently involve cyclical motions of swinging joints, Euler integration without damping induces system instability and gain of energy. In order to simulate pendulums of more than four or five links, we need to introduce a lot of damping into the articulated system, resulting in less plausible motion.

One solution that would benefit the simulation is an adaptive timestep [23]. This method would allow the integrator to vary the stepsize to meet the current accuracy and stability requirements of the simulation.

5. Framework Extensibility

To see how easily the framework can be extended, we implemented some additional behavior beyond rigid body and articulated dynamics. While those two behaviors are also implementing the interfaces of the framework, the framework itself does not know about the the state setup, derivatives, and simulation information. After implementing that information, however, we can quickly define new behaviors.

We have extended the rigid body class in several ways to get new behaviors. One simple idea was to make a rigid body that does not spin. We accomplished that by overriding `AddForce()`, `ApplyImpulse()`, and `ApplyTestImpulse()` methods. We changed the functions so that the forces and impulses act only on the center of mass.

A more complicated example was flocking behavior. We extended the `RigidBody`

class into a `Boid` class that behaves according to basic flocking rules as described by Reynolds [24]. A simple super-class `Flock` was also created to add some higher-level functions for managing the flock. The header for the `Flock` class is presented below.

```
class Flock : public Object {
public:
  Flock();
  void AddBoid(Boid *b);
  void RemoveBoid(Boid *b);

  void SetKaKvKc(const double &Ka, const double &Kv, const double &Kc);
  void SetDistAngles(const double &innerDist, const double &outerDist,
                     const double &innerAngle, const double &outerAngle);
  void SetShader(const string &shader);

  //Object
  bool ImplementsInterface(const InterfaceType &type, void **iObj);
protected:
  vector<Boid*> boids;
};
```

The class defines methods for adding and removing boids to the flock. Additionally, it has methods for setting shading properties and animation parameters for the boids. Finally, it defines methods that a child of an `Object` class must implement. The `Boid` header is given below. As can be seen, the header is very short as a lot of the information is reused from the rigid body definition:

```
class Boid : public RigidBody {
  friend class Flock;
public:
  Boid(Engine *e, string set, string model);

  //redefining some functions
  void AccumulateForces();
  void UpdateState(const SimulationStateState &state);
protected:
  double Ka, Kv, Kc;
  double desiredVelocity, maxForce;
  double visibilityDistance, innerVisibilityDistance;
  double visibilityAngle, innerVisibilityAngle;
  vector <Boid*> *others;
};
```

The boid only needs to redefine some of the integration functions. The primary behavior comes from the `AccumulateForces()` and `UpdateState()` functions. Additionally, the boid needs to define a number of useful parameters. `Ka`, `Kv`, and `Kc` are

animation parameters for coefficients of avoidance, velocity matching, and clustering respectively. `desiredVelocity` is the optimum velocity for the boid and `maxForce` is a limit on how much force the boid can experience due to flocking. The visibility parameters control the distance and angles at which the boid responds to its neighbors.

The `AccumulateForces()` function is responsible for the flocking behavior. First, it determines which neighbors are visible to each boid based on the distance and angle parameters. Then, it collects some distance and vector information from each visible neighbor. The forces due to each of the three behaviors (avoidance, velocity matching, and clustering) are then computed. Finally, the final force is computed based on the `maxForce` parameter. To get basic flocking, the entire function is less than 50 lines. The body of `AccumulateForces()` as well as the rest of the `Boid` class is presented in Appendix A.

`UpdateState()` performs an auxiliary role and determines the orientation of the boid depending on its velocity. With just these functions, we quickly get a new dynamic behavior from the system. We show an example of the flock in action in Figure 26.

We can efficiently develop new functionality for the boid: for example, we can have the boid implement a `Camera` interface to allow tracking or "bird's view". While the basic method simply rotates the boid to face the direction it is moving in, a more elaborate scheme would use torques to rotate the boid. With torques and forces affecting the position and orientation of the boid, the boid would also be able to respond correctly to rigid body collisions. Overriding the `Intersect()` method defined by the `Collidable` interface would allow us to perform other tasks upon collision. We could have the boid bounce, explode, or perform another action. Finally, we could have extended the `KSkeleton` class and had the boid perform a flying animation by applying

Fig. 26. Flocking behavior from the `Boid` class.

actuator forces or simply playing a pre-recorded animation of angles for joints.

We believe that the ease with which we were able to add this additional behavior is indicative of the framework's extensibility. The framework takes care of the more mundane aspects such as adding the object to appropriate interface lists, allowing the developer to focus on more important functionality. For the `Boid` class, we needed to redefine only a few functions and add a new class that essentially holds a list of all the boids in the flock. Building the `Boid` class from scratch would have been more involved, but the developer would have more control of what functionality to include.

C. Dynamics in the Cave

We have integrated the framework into the immersive rendering engine, GUPPY3D. As discussed in section IV.C, the integration was fairly straightforward. After making the necessary adjustments to the immersive engine, we could load dynamic scenes

and display the simulation on multiple facets. Figure 27 shows the flock simulation running in GUPPY3D. Other scenarios, such as billiards and bowling in Figures 28 and 29 were also functional with the dynamics mostly intact.

Fig. 27. Flocks in GUPPY3D.

User, camera, and dynamic information is transferred to the clients over the network using the physics engine's `NetSync` interface. The synchronization ensures that all the information is consistent across the machines. Dynamic objects move coherently from one facet to the next. Figure 30 shows an example of some rigid bodies spanning multiple facets. The rigid bodies overlapping more than one facet have the correct orientation and position. Motion of more complex dynamics, such as the chain in Figure 31, also transfers correctly.

In the simple scenarios that we tested, the speed of the dynamics has not been an issue. The immersive engine easily maintained real-time framerates of about 60 frames per second. Of course, complex dynamic structures and multiple rigid bodies slow

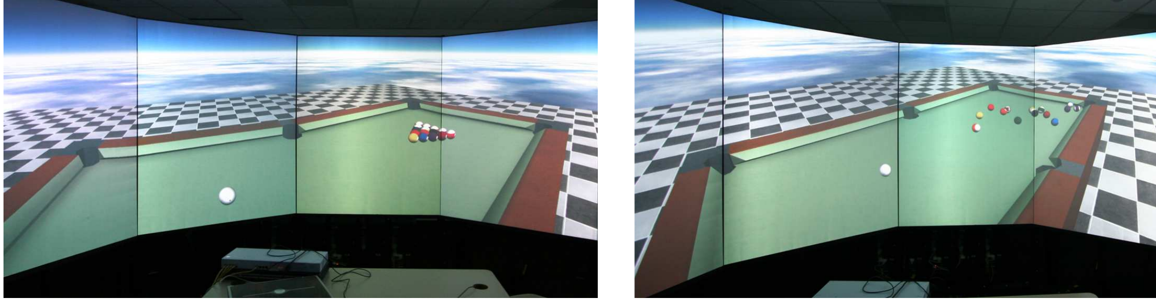

Fig. 28. Billiards in GUPPY3D.

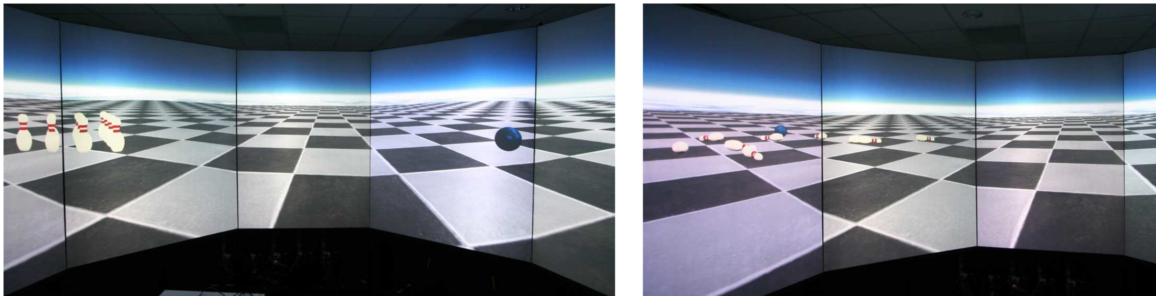

Fig. 29. Bowling in GUPPY3D.

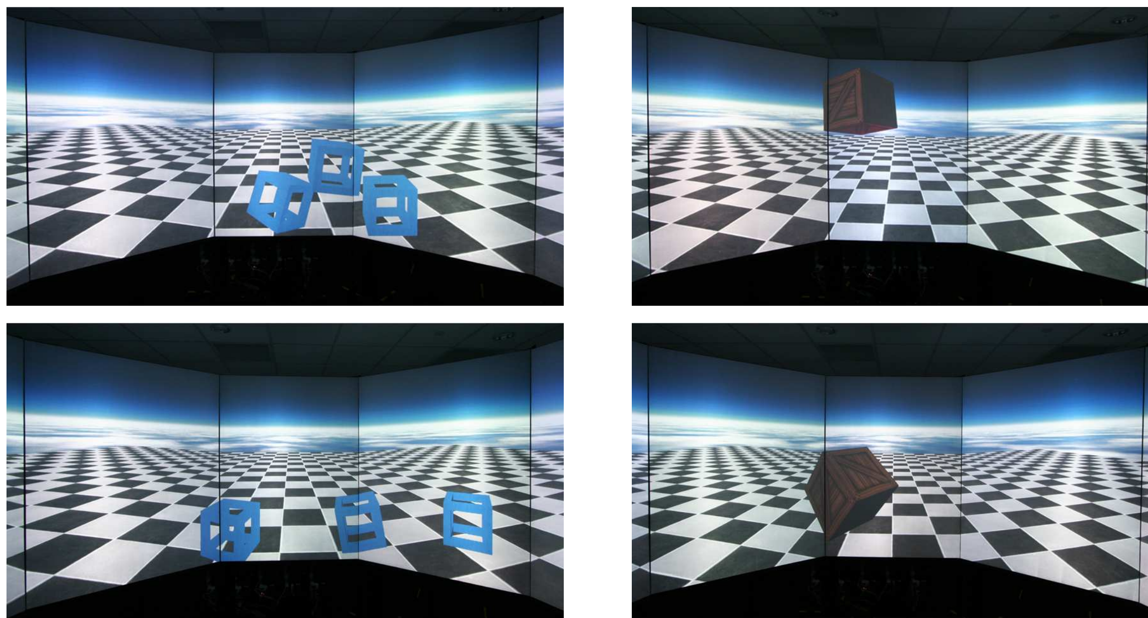

Fig. 30. Rigid bodies in GUPPY3D.

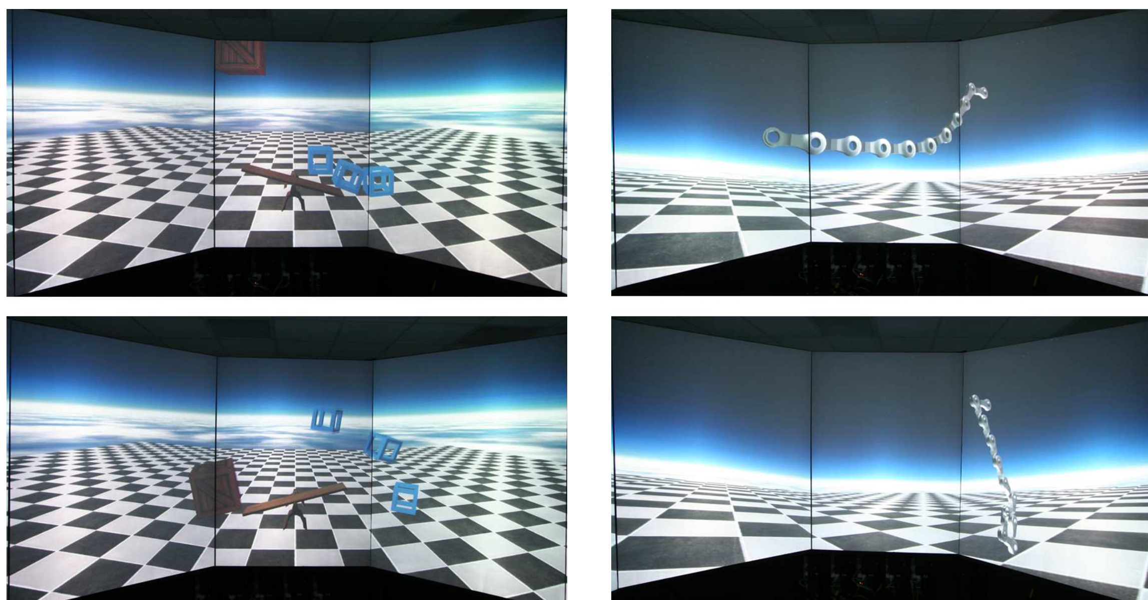

Fig. 31. Articulated dynamics in GUPPY3D.

down the system. The speed issues result mainly from the problems discussed in the previous section of this chapter. The same situations that would cause the standalone system to slow down (such as multiple contacting pins with friction) would slow down the immersive environment equally. Although the framerates dropped as low as 10 frames per second occasionally, the system was still usable. As we mentioned before, there are a number of speedups that could easily bring those numbers to real-time range.

CHAPTER VI

CONCLUSION AND FUTURE WORK

We have designed an extensible framework for simulating physical behaviors. Our implementation supports rigid bodies and articulated dynamics with collision detection, response, resting, and friction. Rigid bodies bounce, roll, and come to rest appropriately. Links of articulated bodies collide with rigid bodies, other links, and come to rest correctly on planes. During the development of the framework, we carefully studied the literature dealing with these concepts. This thesis presents a thorough discussion on the basics of these ideas, as well as illuminating less exposed implementation details.

Our framework can handle any closed object as a rigid body or a link in an articulated structure. Using sphere-trees, we can perform quick intersection tests on polyhedral surfaces and get approximate contact information. Generalized collision response methods can determine the impulses and forces needed to prevent objects from interpenetrating. Together with a generic integration scheme, these methods coherently bring rigid body and articulated dynamic behaviors together. We explain how the collision response is implemented and what functions the interfaces require. Using these interfaces, many other dynamic behaviors can be added to the framework.

We can easily extend existing behaviors to get new ones. We demonstrated that by designing a flocking `Boid` class extended from a rigid body. This extension was quick and efficient. Although we went for simplicity, with a little more work, the framework would allow us to add more complex functionality such as boid animation via articulated dynamics or physically based flight.

Overall, the dynamics added new depth to even the simplest scenes in the immersive environment. While the scenarios we have shown are not complex, they display

the basic capabilities of the framework. More immersive scenes and environments can be designed that take advantage of the physics.

During development, we focused on the functionality of the framework. Therefore, there are a number of issues that need to be resolved. In order to support more dynamic objects, various speedups need to be implemented as mentioned in chapter IV. These include adaptive timestepping, higher order integration, sweeping methods for collision detection, and general speedups throughout the framework. Such improvements would also allow for more complex scenarios and better articulated motion.

Further dynamic features can be implemented to extend the capabilities of the framework. Particle dynamics can provide a number of useful effects such as sparks, smoke, fire, water, explosions, and the like. Computations that take advantage of specialized graphics or physics processors can drastically speed up computations and provide the user with the ability to simulate thousands of dynamic objects in real-time. Additional constraints are also necessary for more elaborate structures. For example, point constraints could be used to introduce loops into articulated structures. Such structures could then be used to physically represent motors, steam engines, and other automated machines.

The framework presented in this thesis is a strong foundation for a generalized physics engine. The system is modular, extensible, and its implementation and concepts are well documented. We have shown how we designed the framework from the ground up as well how we implemented the various behaviors. This system can be used in a standalone form or as an augmentation to an immersive environment and supports a number of different dynamic scenarios. It has the potential to support numerous behaviors, scenes, and environments limited only by the user's imagination.

REFERENCES

[1] S. S. Fisher, M. McGreevy, J. Humphries, and W. Robinett, "Virtual environment display system," in *Symposium on Interactive 3D Graphics*, 1986, pp. 77–97.

[2] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, "Surround-screen projection-based virtual reality: The design and implementation of the cave," in *International Conference on Computer Graphics and Interactive Techniques*, 1993, pp. 135–142.

[3] F. I. Parke, "Next generation immersive visualization enviroments," in *SIGRADI 2002*, Caracas, Venezuela, May 2002, pp. 163–166.

[4] D. Baraff, A. Witkin, and M. Kass, "Physically based modeling," SIGGRAPH '99 Course Notes, 1999.

[5] V. Kokkevis, "Practical physics for articulated characters," in *Game Developers Conference*, San Jose, CA, 2004.

[6] R. Featherstone and D. Orin, "Robot dynamics: Equations and algorithms," in *IEEE International Conference on Robotics and Automation*, San Francisco, CA, 2000, pp. 826–834.

[7] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul, "On-line compuational scheme for mechanical manipulators," *Trans. ASME, Journal of Dynamic Systems, Measurement and Control*, vol. 102, pp. 69–76, 1989.

[8] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *International Journal of Robotics*, vol. 2, pp. 13–29, 1987.

[9] B. Mirtich, "Impulse-based dynamic simulation of rigid body systems," Ph.D. dissertation, University of California at Berkeley, 1996.

[10] S. Gottschalk, M. Lin, and D. Monacha, "Obb-tree: A hierarchical structure for rapid interference detection," in *Proceedings of ACM SIGGRAPH '96*, 1996, pp. 171–180.

[11] T. C. Hudson, M. C. Lin, J. Cohen, S. Gottschalk, and D. Manocha, "V-collide: accelerated collision detection for vrml," in *3D Technologies for the World Wide Web, Proceedings of the Second Symposium on Virtual Reality Modeling Language*, Monterey, CA, 1997, pp. 119–125.

[12] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking," in *Proceedings of ACM SIGGRAPH 2003*, 2003, pp. 871–878.

[13] J. Sethian, "A fast marching level set method for monotonically advancing fronts," in *Proceedings of the National Acadademy of Sciences*, 1996, vol. 93, pp. 1591–1595.

[14] R. W. Cottle, J.-S. Pang, and R. E. Stone, *The Linear Complementarity Problem*, San Diego, CA: Academic Press, 1992.

[15] D. Eberly, *Game Physics*, San Francisco, CA: Morgan Kaufmann, 2004.

[16] C. D. Anderson, "3d engine for immersive virtual environments," M.S. thesis, Texas A&M University, 2003.

[17] N. Llopis, "Programming with abstract interfaces," in *Game Programming Gems 2*, Mark DeLoura, Ed., Rockland, MA: Charles River Media, 2001.

[18] D. Eberly, "Geometric tools," 2007, http://www.geometrictools.com/index.html.

[19] G. Bradshaw and C. O'Sullivan, "Adaptive medial-axis approximation for sphere-tree construction," *ACM Transactions on Graphics*, 2004, pp. 1–26.

[20] G. Bradshaw, "Sphere-tree construction toolkit," 2003, http://isg.cs.tcd.ie/spheretree/.

[21] D. Baraff, "Curved surfaces and coherence for non-penetrating rigid body simulation," *Computer Graphics*, vol. 24, no. 4, pp. 19–28, 1990.

[22] P. G. Kry and D. K. Pai, "Continuous contact simulation for smooth surfaces," *ACM Transactions on Graphics*, 2003, vol. 22, pp. 106–129.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, New York, NY: Cambridge University Press, 1992.

[24] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," *Computer Graphics*, vol. 21, no. 4, pp. 25–34, 1987.

APPENDIX A

CODE FOR BOID CLASS

This appendix defines the `AccumulateForces()` and `UpdateState()` function of the `Boid` class.

```
void Boid::UpdateState(const SimulationStateState &state){
  RigidBody::UpdateState(state);
  //update rotation based on velocity
  if(velocity.norm() > SMALLNUMBER){
    //find rotation quaternion
    Vector3d cross(Vector3d(1, 0, 0) % velocity.normalize());
    nState->rotation = Quaternion(RAD2DEG *
      acos(Vector3d(1, 0, 0) *
      velocity.normalize()), cross.x, cross.y, cross.z);
    Vector3d z(nState->rotation * Vector3d(0, 0, 1));
    double turnMagnitude = z * nState->force / velocity.norm();
    if(turnMagnitude > 1) turnMagnitude = 1;
    if(turnMagnitude < -1) turnMagnitude = -1;
    nState->rotation = nState->rotation *
      Quaternion(turnMagnitude * 90, 0, 0, 1);
  }
}

void Boid::AccumulateForces(){
  RigidBody::AccumulateForces();
  vector<Boid*> nearBoids;
  vector<double> nearWeights;
  vector<double> dists;
  vector<Vector3d> Us;
  //check based on distance first
  for(unsigned i = 0; i < others->size(); i++){
    if((*others)[i] == this)
      continue;
    Vector3d otherPos((*others)[i]->GetPosition());
    //check distance
    double dist = (otherPos - GetPosition()).norm();
    if(dist < visibilityDistance){
      double weight = dist < innerVisibilityDistance ? 1.0 :
        (visibilityDistance - dist) /
        (visibilityDistance - innerVisibilityDistance);
      //check angle
      Vector3d U = (otherPos - GetPosition()).normalize();
      Vector3d v(nState->rotation * Vector3d(1, 0, 0));
      double theta = acos(U.normalize() * v.normalize()) * RAD2DEG;
      if(theta < visibilityAngle){
        weight *= theta < innerVisibilityAngle ? 1.0 :
          (visibilityAngle - theta) /
          (visibilityAngle - innerVisibilityAngle);
        nearBoids.push_back((*others)[i]);
        nearWeights.push_back(weight);
        dists.push_back(dist);
```

```
        Us.push_back(U);
      }
    }
  }

  //forces for avoidance, velocity matching, and clustering
  Vector3d Fa, Fv, Fc;
  for(unsigned i = 0; i < nearBoids.size(); i++){
    Fa += mass * (-nearWeights[i] * Ka * (1.0 / dists[i]) * Us[i]);
    Fv += mass * ( nearWeights[i] * Kv *
      (nearBoids[i]->GetVelocity() - GetVelocity()));
    Fc += mass * ( nearWeights[i] * Kc * dists[i] * Us[i]);
  }

  //prioritize forces
  Vector3d newForce;
  double magFa = Fa.norm();
  if(magFa > maxForce){
    newForce = Fa.normalize() * maxForce;
  }
  else {
    newForce = Fa;
    if((Fa + Fv).norm() < maxForce){
      newForce += Fv;
      if((Fa + Fv + Fc).norm() < maxForce){
        newForce += Fc;
      }
      else
        newForce = maxForce * (Fa + Fv + Fc).normalize();
    }
    else
      newForce = maxForce * (Fa + Fv).normalize();
  }
  double dist = GetPosition().norm();
  if(dist > 300)
    newForce += (dist - 300) * -GetPosition().normalize();
  AddForce(ZeroVector, newForce);
}
```

VITA

Name:

Alexander Nikolai Timchenko

Address:

Department of Architecture

Langford C418

Texas A&M University

3137 TAMU

College Station, TX 77840-3137

Email:

alex@viz.tamu.edu

Education:

B.S., Computer Science, Texas A&M University, 2004