Summarizing the Physics Engine

Congratulations! You have learned the basic ideas and concepts behind and completed the implementation of a 2D physics engine. This chapter will summarize all of your work done from Chapter 1 to 4, what you should understand and take away from this book, and highlight improvements or future explorations on the physics engine that you have created.

This chapter begins by summarizing all of the physics engine theories and concepts that you have learned and used throughout the book. Next, a detailed list of source code files, and the associated functions that you have written are presented serving as a simple ‘readme’ file. Lastly, further topics you can explore and possibly implemented in your physics engine will be presented as a starting point for your future endeavors with game physics engines. This chapter will also include a simple project serving as the final and complete functionality and features testing of your engine. You can follow the project guide on setting up and running the simulation, or be creative and set up your own test cases.

# The Concepts and Theories

This book is design to guide you to build your own physics simulation. As such, all topics introduced relates to the building of such a system.

* **Rigid Shape** - A primitive that does not change its shape during physical interaction. In order to support efficient interaction simulation, these are usually simple geometric shapes, e.g., circles & rectangles. A rigid shape has its own attributes that support physics simulation such as mass, width, height, center of gravity, inertia, friction, restitution. etc.
* **Engine Loop** - A continuous running loop that updates the object states, invokes the calculations of inter-object interactions, and renders the objects. The engine loop must cycles through all operations and maintains a real-time performance. By implementing a fixed time step update in the loop, it becomes straightforward to simulate movement integration and maintain a deterministic game state.
* **Collision Detection** - An algorithm to determine if objects have overlapped and/or interpenetrated objects.
* **Broad Phase Method** - An optimization for collision detection by exploiting the proximity of objects. Axis-aligned bounding boxes are used by the engine to reduce the overhead of invoking actual collision detection algorithms.
* **Separating Axis Theorem** - One of the most popular algorithms for detecting collisions between general convex shapes in 2D. It is typically preceded with an initial pass of a broad phase method to improve its overall performance. This algorithm can detect collisions between axis-aligned as well as rotated shapes.
* **Collision Information** - The information describing the details of a collision including, interpenetration depth, normal direction the caused the interpenetration, begin and end of an interpenetration. This information is essential for resolving a collision.
* **Symplectic Euler Integration** - A method of approximating integrals based on initial values. This engine uses the Symplectic Euler Integration to approximate an object’s new linear and rotational velocities, and its new position.
* **Positional Correction** - The process of separating two interpenetration objects using collision information collected during collision detection.
* **Relaxation Loop** - An iterative loop in the core of the physics engine that repeatedly and incrementally apply positional correction on interpenetrating objects in an attempt to remove the occurrence of colliding object interpenetration.
* **Impulse Method -** A largely simplified physically based collision response formulation that is capable of capturing object bounciness and friction considerations during a collision.
* **Collision Resolution** - A process that determines how objects should respond after a collision. When applying the Impulse Method to resolve a collision, colliding objects receive new linear and angular velocities.

# The Engine Source Code

The following are the list of source code files and the associated functionality.

* Core.js

Core engine loop

Update function

Drawing function

UI control

* Physics.js

Collision detection

Relaxation loop

Positional correction

Resolving collision

* CollisionInfo.js

Collision information object

Constructor and getter/setter

* Vec2.js

2D vector calculation

* RigidShape.js

Base class of rigid shape

Constructor

Update function

Bounding box collision test for broad phase method support

* Rectangle.js & Circle.js

Inherit from rigid shape base class

Specific constructor for each

Rotate function

Draw function

Move function

* Rectangle\_collision.js & Circle\_collision.js

Collision detection functions

Gather collision information

* UserControl.js

User input controller

* MyGame.js

Simulation scene controller

* Index.html

Script calling

Initialize simulation scene

# The Cool Demo Project

This project guides you in setting up the scene to test the functionality of your physics engine implementation. You can see an example of this project running in Figure 5-1. The source code to this project is defined in the A Cool Demo Project folder.

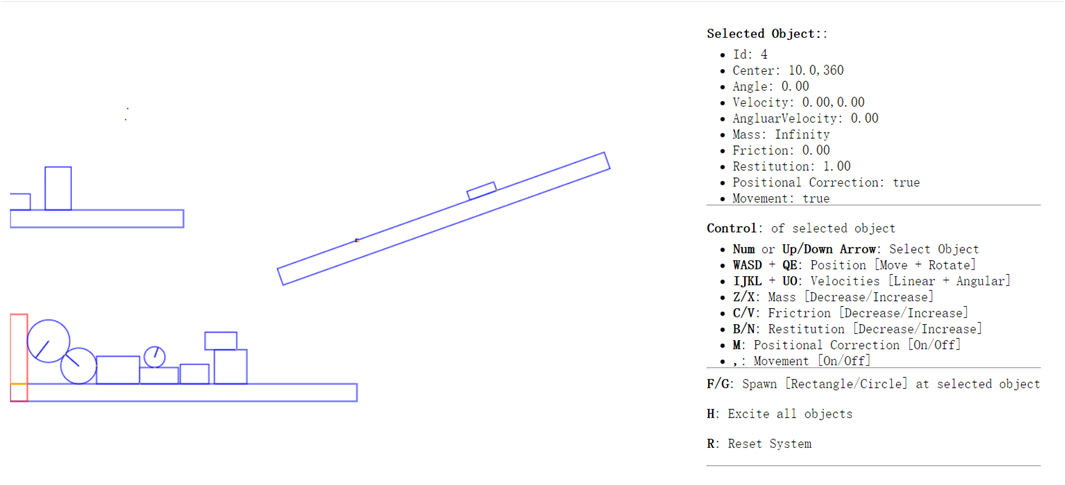


Figure 5-1. Running The Cool Demo Project.

Project Goal:

* To test and engage with all the functionalities and features of the physics engine

## Modify Simulation Scene

Let’s start by modifying the simulation scene:

1. Edit the MyGame.js file.
2. Replace all the code inside the MyGame constructor to create a new scene for the simulation.

"use strict";

/\* global height, width, gEngine \*/

function MyGame() {

}

1. In the MyGame constructor, create four platforms with one rotated for testing the angular movements.

//...continue from previous step

var r1 = new Rectangle(new Vec2(500, 200), 400, 20, 0, 0.3, 0);

r1.rotate(2.8);

var r2 = new Rectangle(new Vec2(200, 400), 400, 20, 0, 1, 0.5);

var r3 = new Rectangle(new Vec2(100, 200), 200, 20, 0);

var r4 = new Rectangle(new Vec2(10, 360), 20, 100, 0, 0, 1);

//...more in next step

1. Create 10 circle and rectangle objects with random attributes to begin with the simulation.

//...continue from previous step

for (var i = 0; i < 10; i++) {

var r1 = new Rectangle(new Vec2(Math.random() \* gEngine.Core.mWidth,

Math.random() \* gEngine.Core.mHeight / 2),

Math.random() \* 50 + 10, Math.random() \* 50 + 10,

Math.random() \* 30, Math.random(), Math.random());

r1.mVelocity = new Vec2(Math.random() \* 60 - 30, Math.random() \* 60 - 30);

r1 = new Circle(new Vec2(Math.random() \* gEngine.Core.mWidth,

Math.random() \* gEngine.Core.mHeight / 2),

Math.random() \* 20 + 10, Math.random() \* 30,

Math.random(), Math.random());

r1.mVelocity = new Vec2(Math.random() \* 60 - 30, Math.random() \* 60 - 30);

}

## Observation

You can see now that there are no borders in the scene anymore. This allows objects to fall off the screen and not crowd the space. In this way you can continue to create new objects and observe the simulation of object behaviors. You can also test the performance of your engine by creating more objects at the beginning of the simulation. Note that this book provides you with the basic understanding of creating your own physics engine. There are plenty of rooms for improvements, ranging from choosing alternative algorithms, supporting different features, to optimizing the efficiency of the calculations, etc. The next section will point out some of the topics you could look into to improve your engine.

# Further Exploration and Related Topics

With your physics engine now complete you may be asking yourself, what now? How should I proceed with the knowledge I have gained, what should I do with the physics engine I created or what should I learn next? Ultimately, as is most often the case, the answer is that, it depends. It depends on your interests in game physics engines in the first place and why you decided to read and follow along with this book. If your desire was to create a game or game engine from scratch, you may wish to integrate this physics engine into your own game engine or an existing game engine in order to add its rigid body physics functionality to the project. If your reason had a more academic nature with the goal of learning and understanding how game physics engines function you may want to explore further into related topics within game physics.

Regardless of which category you lie in you may wish to extend the functionality of the physics engine by improving its performance and capabilities by adding more advanced features or components. If that is the case, then the following topics provide you with some suggested jumping off points for further exploration within game physics.

## Physics Topics

* **Advanced 2-D Rigid Body Physics** - If you enjoyed the Impulse Method approach and are looking to improve the functionality of your physics engine by adding features such as kinematics (often used for moving platforms), joints (from more complex rigid body behavior) or a host of other great features we suggest that you look the Box2D physics engine and the literature from its creator Erin Catto. Box2D is the game physics engine that popularized the Impulse Method and is available in several programming languages. <http://box2d.org/>
* **Verlet Physics** - If you’re looking to simulate soft body physics then we suggest exploring Verlet physics. Verlet physics provides a fast and simple way to simulate soft bodies, such as ragdolls, ropes, jelly like objects and even cloth through the use of particles, constraints (springs) and Verlet Integration to build complex soft bodied objects.  In particular, we suggest you take a look at Thomas Jakobsen’s paper on *Advanced Character Physics* which is probably the most popular starting point for people interested in game physics due to its ease of implementation and understandability. The downside of Verlet physics is the potential of instability when applied to rigid body simulations.
* **Networked Physics** - The subject of networked physics contains its own unique set of problems that need to be addressed many of which revolve around synchronization. To get your bearings on the subject we suggest you take a look at the following website. <http://gafferongames.com/game-physics/>
* **3-D Rigid Body Physics** - If you’re interested in venturing into 3-D physics simulations a great starting point is the Impulse Method! The great thing about the Impulse Method is that it can also be used in 3-D physics as well as 2-D. Newcastle University provides some great information on implementing the impulse method in 3-D. <https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/>

## Collision Detection Topics

* **Continuous Collision** - Continuous collision is a method to solve the problem of physics objects passing through other physics object geometries that are too small or traveling at too high of velocities. This is a problem due to the discrete time step nature of game engines. There are several approaches in order to solve this problem. A great place to start and get your bearings for the topic is Erin Catto’s GDC (Game Developers Conference) presentation. <http://www.gdcvault.com/play/1018239/Physics-for-Game-Programmers-Continuous>
* **Collision Callbacks** - Collision callbacks provide a more advanced and flexible collision behavior. They can be used to customize the behavior for your physics objects such as OnCollisionEnter or OnCollisionExit. In addition, they can also be useful for passing any collision information needed for any game logic. Collision callbacks are often a key feature for more advanced physics engines.
* **GJK Collision Detection** – The GJK (Gilbert–Johnson–Keerthi) algorithm is an alternate collision detection method to the Separating Axis Theorem. GJK provides more flexibility and performs collision detection for many sided convex polygons.
* **Spatial Partitioning** - Spatial partitioning is a more advanced broad phase method commonly used in physics engines in order to improve performance for both collision detection and response. The method divides up the world space objects into discrete areas in order to detect likely collisions. One of the more commonly used spatial partitioning techniques in 2D is known as Quadtree.

# Reference

The following are some of the references we consulted when we learned this topic.

* General definitions: <https://en.wikipedia.org/>
* Physic Shape and attributes: <http://buildnewgames.com/gamephysics/>
* Separating Axis Theorem: <http://www.metanetsoftware.com/technique/tutorialA.html#section3>
* Resolve collision without rotation: <https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-friction-scene-and-jump-table--gamedev-7756>
* The formula of impulse in collision with rotation: <http://www.myphysicslab.com/collision.html>
* Resolve collision rotation and Separating Axis Theorem: <https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-oriented-rigid-bodies--gamedev-8032>