Chapter 11
Collision Detection Using the GJK Algorithm

William N. Bittle
dyn4j.org, USA

ABSTRACT

GJK is a fast and elegant collision detection algorithm. Originally designed to determine the distance between two convex shapes, it has been adapted to collision detection, continuous collision detection, and ray casting. Its versatility, speed, and compactness have allowed GJK to be one of the top choices of collision detection algorithms in a number of fields.

INTRODUCTION

Collision detection is the process of detecting when two objects intersect, overlap, or collide. It's found in simulations, video games, robotics, and even basic GUI applications. The mouse, a standard user interface device, uses a simplified form of collision detection to achieve tasks like issuing a click command on a button or moving a window. The mouse's current position must be found inside the rectangle that forms the bounds of the button or window to perform the respective action. This type of collision detection is often referred to as hit-testing.

DOI: 10.4018/978-1-4666-1634-9.ch011

On the other hand, robotics, simulations, and video games have more challenging requirements. For applications like these it's essential to know when a player attempts to exit the playable area, when two objects occupy the same space, or when to avoid a robotic arm crashing through a wall. However, unlike the interaction between the mouse and GUI, objects within these environments can have any shape and can move and rotate freely. Early applications used simple geometric constructs, like circles and non-rotating axis-aligned rectangles to make detection easy. Detecting if two circles are colliding can be accomplished by checking if the distance between their centers is greater than the sum of their radii. In addition, detecting if two non-rotating axis-aligned rect-

angles are colliding can be done by comparing the extents, usually the top-left and bottom-right vertices, against each other with simple less-than greater-than comparisons. Yet, as technology has advanced, objects within the environment have become more complex, requiring the use of less specialized collision detection routines. One solution might be to use a separate collision detection algorithm for each shape type pair; circle-circle, circle-axis-aligned rectangle, and so on. This can be a good solution for applications that require only a few different shape types, but it can quickly become unmanageable as the number of shape types increase. An ideal solution would be able to handle any moving/rotating shape efficiently and accurately with one algorithm. The GJK algorithm is one such solution.

In addition to the increased complexity, the size of the environments themselves has increased. It's unlikely to find an environment that includes only a handful of objects. Likewise, it's difficult to find objects that are represented with only *one* simple shape. As such, collision detection software has evolved into a phased approach, usually including two or three phases, which attempt to reduce the number of collision tests that must be performed by expensive algorithms. The first phase is referred to as the *broad-phase.* The broad-phase performs the n x n collision tests, where n is the number of objects in the scene, using one simple geometric shape that encloses the entire object. This phase is designed to be extremely fast but at the same time conservative; meaning it should not miss any collisions, but may report false collisions. Sweep and Prune, BSP Trees, Uniform Grids, and Hierarchical Grids are some examples of broad-phase collision detection algorithms. The pairs found in the broad-phase are then passed to either an intermediate phase, called the *mid-phase*, or directly onto the *narrow-phase*. The mid-phase is used to reduce the number of collision tests for objects that are represented by a combination of simple shapes, often using the same algorithms as the broad-phase or variations thereof. Finally, the narrow-phase uses the exact geometry of the objects to perform an exact collision test. The narrow-phase is typically the most expensive algorithm. GJK and the Separating Axis Theorem are some examples of narrow-phase collision detection algorithms. The phased approach is not required but can improve performance significantly. This chapter focuses solely on the GJK algorithm.

Beginning with some background information we will cover the basic principles of the algorithm along with common pitfalls using concrete examples and pseudo code. After which, we will cover a simplified version of the original method for obtaining the distance between two convex shapes and the closest points. Next, we will cover a supplementary algorithm to obtain collision information useful in collision resolution. Finally, we will touch on how the algorithm has been applied to continuous collision detection and ray casting.

BACKGROUND

To begin detecting collisions between objects within an environment, they must have a representation of their shape. The most common shapes used are circles, rectangles, and polygons because of their simplicity. In three dimensions common shapes include spheres, rectangular boxes, and cylinders. An object, like a chair, can be represented by one or more of these simple shapes depending on the accuracy required. The closer the simple shapes fit to the original object the higher the accuracy but the lower the performance. We could use a rectangular box that encloses the entirety of the chair or we could create multiple rectangular boxes that enclose each leg, arm, and other features. Either way, the representation of these simple shapes is required for a collision detection algorithm. For the remainder of the chapter we will focus on two types of shapes; circles and