Collision Detection

Chapter 11

Content

1. Collision Basics

- **Deciding what collisions to test.** It seems kind of ridiculous to test whether two objects collide if they are on opposite sides of the world. Also, a world with 1,000 moving objects would require you to test each object against every other object, or 999,000 tests in all. So, you should try to limit the number of objects you need to test as much as possible. An easy way to do this is only test objects that are close.

- **Detecting a collision.** Collision detection really depends on how accurate you want the collisions to be. You could provide perfect collision detection and test every polygon in one object with every polygon in another object, but just imagine the amount of computation involved. Likewise, for the 2D world, you could test every pixel from one sprite to every pixel from another sprite. Usually, games settle with collision-detection techniques that are slightly inaccurate but can be processed quickly.

- **Handling a collision.** If an object collides with something, the collision will be handled in different ways depending on the type of collision. For example, a projectile colliding with a robot might destroy both the projectile and the robot. An object bumping against a wall might slide against the wall. And so on.

Collision Basics (2)

- Eliminate as many collision tests as possible.
- Quickly decide whether there is a collision.
- Provide collision detection that is accurate enough for the game.
- Handle the collision in a way that doesn't distract the user from the game.

Collision Basics (3)

- Consider that everything that moves in the game is an object, whether it's a monster, the player, a projectile, or whatever. For each object, follow these steps:
  - Update the object's location.
  - Check for any collisions with other objects or with the environment.
  - If a collision is found, revert the object to its previous location.
- Note: you check for a collision after each object moves. Alternatively, you could just move all the objects first and then check collisions afterward. However, you'd have to store the object's previous location with the object, and problems could occur if three or more objects collide.
- Also, this basic algorithm just reverts an object to its previous location if a collision occurred. Normally, you want to use other types of collision handling based on the type of collision.

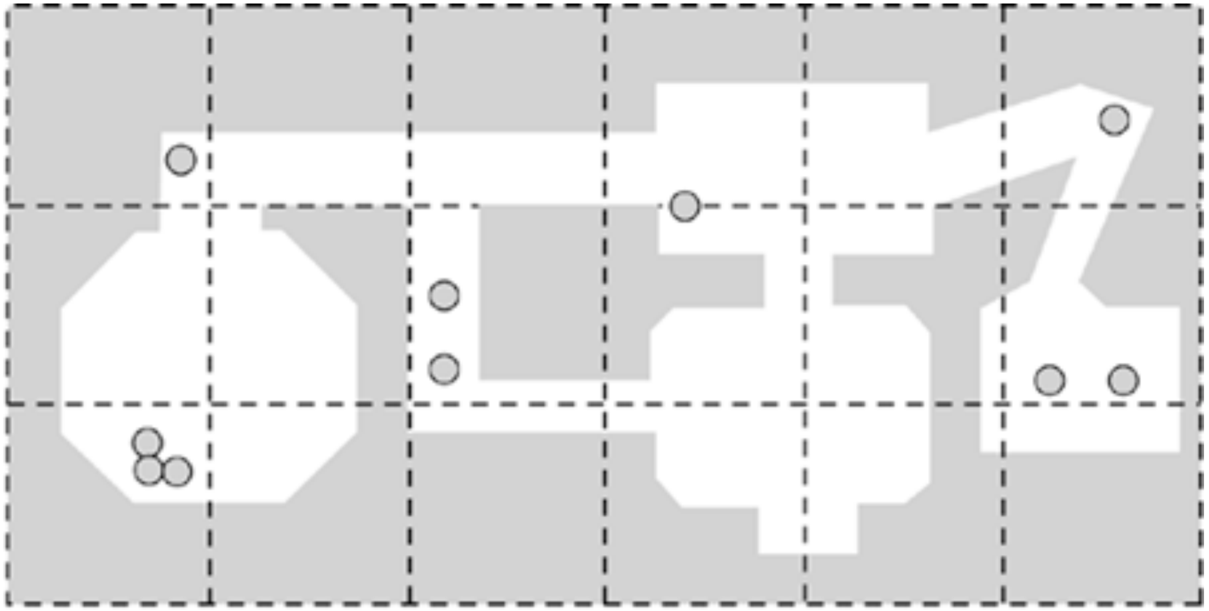
2. Object-to-Object Collisions

- Ideally, no matter how accurate you want the collision detection to be, it's a good idea to both eliminate as many object-to-object tests as possible and do a few other tests first to ensure that two objects have a good chance of colliding.

- **Eliminating Tests**
  - If an object doesn't move from one frame to the next, you don't need to do any collision tests for that object. For example, a crate that just sits there will never collide with anything. Other objects will collide with the crate, but those collisions are handled by the other objects.
  - To help eliminate collision tests further, you really should test only objects that are in the same proximity.

Object-to-Object Collisions (2)

- **Eliminating Tests (cont.)**
  - One way to do this is to arrange objects on a grid. Each object exists in one cell on the grid. Even though an object's bounds could extend to other cells, the object exists in only one cell. This way, an object needs to test for collisions only with objects in its own cell and its surrounding cells.

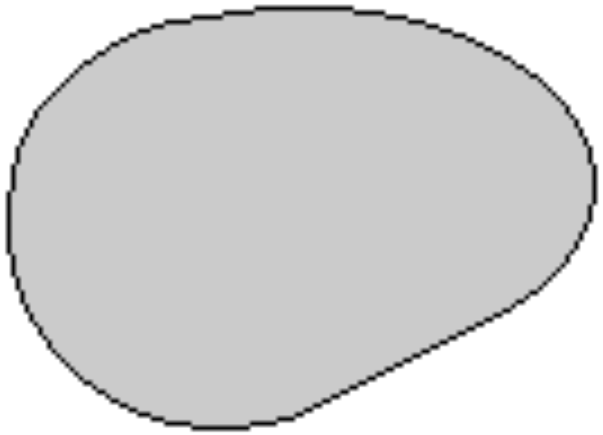


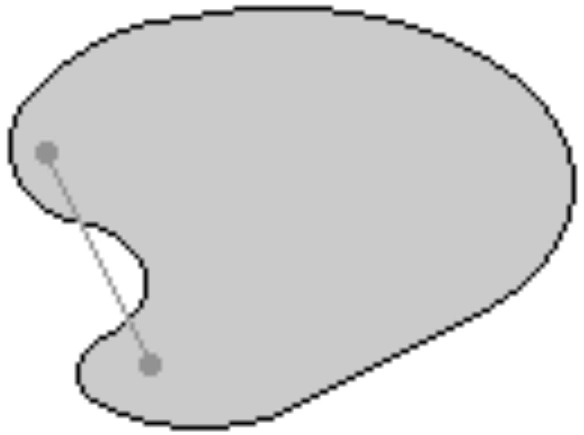
To reduce the number of object-to-object collision tests, you can isolate objects on a grid and only test an object against other objects that are in the same cell and its surrounding cells.

Object-to-Object Collisions (3)

- **Eliminating Tests (cont.)**
  - Other ways to isolate objects include 1D or 3D versions of the grid concept. For example, in a side-scrolling game, objects could be in a list sorted by their x location, so only neighboring objects in the list test for collisions. For a 3D game, objects could be isolated in a 3D grid instead of a 2D grid, so each cell would be a cube instead of a square. In this chapter, for collision with the 3D engine, we use the 2D version.
  - Isolating objects on a grid also has the benefit of easily applying object culling. For example, with a top-down engine, you would draw only the objects in visible cells. For a **BSP** tree, you would draw only the objects in cells that have visible leaves.
  - The code for arranging objects in a grid is trivial and is implemented for this chapter in the **GridGameObjectManager** class. When an object is updated, the **checkObjectCollision()** method is called. This method checks for a collision with any objects in the object's surrounding cells.

Excursus: Convex Hull in Eucledian Geometry


(a)


(b)

convex

non-convex

Object-to-Object Collisions (4)

```
/**
    The Cell class represents a cell in the grid. It contains
    a list of game objects and a visible flag.
*/
private static class Cell {
    List objects;
    boolean visible;

    Cell() {
        objects = new ArrayList();
        visible = false;
    }
}

...

/**
    Checks to see if the specified object collides with any
    other object.
*/
public boolean checkObjectCollision(GameObject object,
    Vector3D oldLocation)
{

    boolean collision = false;

    // use the object's (x,z) position (ground plane)
    int x = convertMapXtoGridX((int)object.getX());
    int y = convertMapYtoGridY((int)object.getZ());

    // check the object's surrounding 9 cells
    for (int i=x-1; i<=x+1; i++) {
        for (int j=y-1; j<=y+1; j++) {
            Cell cell = getCell(i, j);
            if (cell != null) {
                collision |= collisionDetection.checkObject(
                    object, cell.objects, oldLocation);
            }
        }
    }

    return collision;
}
```

Object-to-Object Collisions (5)

- An inaccurate but fast technique for performing collision detection is the use of bounding spheres, such as the one in the Figure.

- **Bounding Spheres**

  - This idea is that if two objects' spheres collide, the collision is treated as if the two objects collide. A first try at testing whether two objects' spheres collide works something like this:

```
dx = objectA.x - objectB.x;
dy = objectA.y - objectB.y;
dz = objectA.z - objectB.z;
minDistance = objectA.radius + objectB.radius
if (Math.sqrt(dx*dx + dy*dy + dz*dz) < minDistance) {
     // collision found
}
```

Object-to-Object Collisions (6)

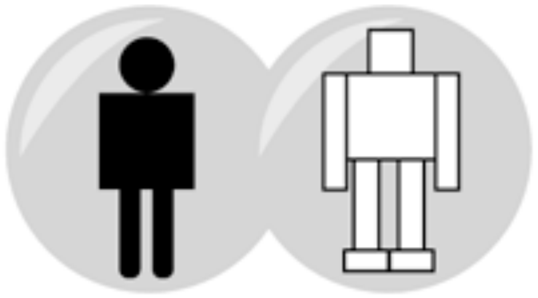
- **Bounding Spheres (cont.)**
  - However, that Math.sqrt() function call involves a lot of computation. Instead, you could square both sides of the equation to get something a bit simpler:

```
if (dx*dx + dy*dy + dz*dz < minDistance*minDistance) {
    // collision found
}
```

  - If your game is in a 2D world instead of 3D, you can test for circles instead of spheres, taking the z coordinate out of the equation.

Object-to-Object Collisions (7)

- **Bounding Spheres (cont.)**
  - Testing bounding spheres is easy, but it's not very accurate. For example, in the figure, the bounding sphere of the player collides with the bounding sphere of the robot, even though the player and the robot don't actually collide.

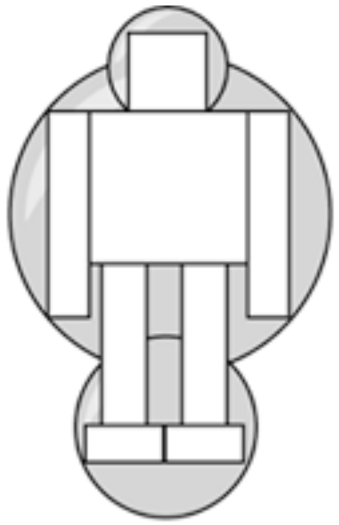


Bounding sphere inaccuracy: The two spheres collide, but the objects don't.

  - Having this amount of inaccuracy is fine for many games. For example, for a fast-moving action game in which you're running around picking up food, you probably won't care whether you pick up the item a little before you actually touch the item. But for other situations, the inaccuracy can be annoying, such as when you get wounded by a creature that you know didn't touch you.

Object-to-Object Collisions (8)

- **Bounding Spheres (cont.)**
  - Another method is to use a second set of spheres as a more accurate set of tests. In this figure, the robot has three bounding spheres that more accurately describe the robot's shape.
  - After the bounding spheres of two objects test positive for a collision, their second set of spheres could be tested. If any of the player's secondary spheres collide with any of the robot's secondary spheres, then a collision occurs.



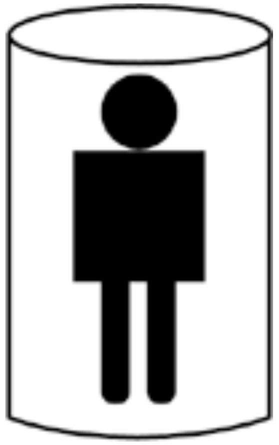
Multiple spheres can be used for more accurate boundary tests.

Object-to-Object Collisions (9)

- **Bounding Spheres (cont.)**
  - **Sphere tree** or **sphere subdivision**. This way, you can quickly reject non-colliding objects and have more accurate tests for potential collisions. This also lets you know which part of your object was hit, allowing you to act accordingly. For example, you could have just your robot's lower leg fall off if a missile strikes it.
  - Note that sphere trees need to rotate with the object as the object rotates. For example, the spheres need to follow a robot's arm as it moves around.
  - To sum it up, for a typical frame, most of the objects require no collision test, some objects require a simple collision test, and a few objects require the more complex, computationally expensive collision tests.

Object-to-Object Collisions (10)

- **Bounding Cylinders**
    - An alternative to bounding spheres is upright bounding cylinders, shown in this figure This reduces collision tests to a 2D circle test and a 1D vertical test.



An upright bounding cylinder can be used for collision detection.

- Upright bounding cylinders tend to describe tall, thin objects (such as players or monsters) more accurately than just one sphere.

Object-to-Object Collisions (11)

- **Bounding Cylinders (cont.)**
  - All the basic collision-detection code in this chapter goes in the **CollisionDetection** class. The methods for handling object-to-object collision.

```
/**
    Checks if the specified object collisions with any other
    object in the specified list.
*/
public boolean checkObject(GameObject objectA, List objects,
    Vector3D oldLocation)
{
    boolean collision = false;
    for (int i=0; i<objects.size(); i++) {
        GameObject objectB = (GameObject)objects.get(i);
        collision |= checkObject(objectA, objectB,
            oldLocation);
    }
    return collision;
}


/**
    Returns true if the two specified objects collide.
    Object A is the moving object, and Object B is the object
    to check. Uses bounding upright cylinders (circular base
    and top) to determine collisions.
*/
public boolean checkObject(GameObject objectA,
    GameObject objectB, Vector3D oldLocation)
{
    // don't collide with self
    if (objectA == objectB) {
        return false;
    }

    PolygonGroupBounds boundsA = objectA.getBounds();
    PolygonGroupBounds boundsB = objectB.getBounds();
```

```
   // first, check y axis collision (assume height is pos)
    float Ay1 = objectA.getY() + boundsA.getBottomHeight();
    float Ay2 = objectA.getY() + boundsA.getTopHeight();
    float By1 = objectB.getY() + boundsB.getBottomHeight();
    float By2 = objectB.getY() + boundsB.getTopHeight();
    if (By2 < Ay1 || By1 > Ay2) {
        return false;
    }

    // next, check 2D, x/z plane collision (circular base)
    float dx = objectA.getX() - objectB.getX();
    float dz = objectA.getZ() - objectB.getZ();
    float minDist = boundsA.getRadius() + boundsB.getRadius();
    float distSq = dx*dx + dz*dz;
    float minDistSq = minDist * minDist;
    if (distSq < minDistSq) {
         return handleObjectCollision(objectA, objectB, distSq,
            minDistSq, oldLocation);
    }
    return false;
}


/**
    Handles an object collision. Object A is the moving
    object, and Object B is the object that Object A collided
    with. For now, just notifies Object A of the collision.
*/
protected boolean handleObjectCollision(GameObject objectA,
    GameObject objectB, float distSq, float minDistSq,
    Vector3D oldLocation)
{
    objectA.notifyObjectCollision(objectB);
    return true;
```

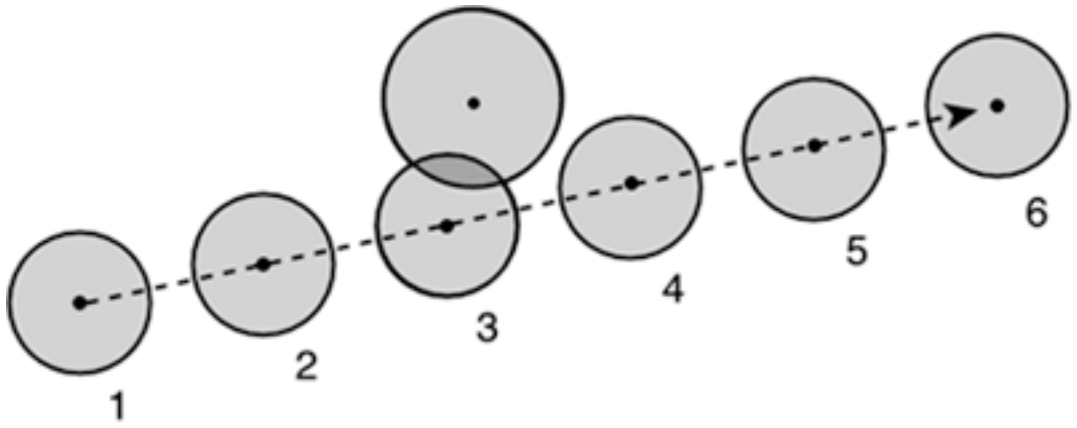
Object-to-Object Collisions (12)

- **Bounding Cylinders (cont.)**
  - The **handleObjectCollision()** method just notifies the moving object that a collision occurred through the **notifyObjectCollision()** method. This and other notify methods exist in the GameObject class, but they don't do anything by default—subclasses of GameObject can override notify methods if they want.
  - For example, in the Blast class, the notifyObjectCollision() method is used to destroy a bot if it collides with one:

```
public void notifyObjectCollision(GameObject object) {
    // destroy bots and itself
    if (object instanceof Bot) {
        setState(object, STATE_DESTROYED);
        setState(STATE_DESTROYED);
    }
}
```

Object-to-Object Collisions (13)

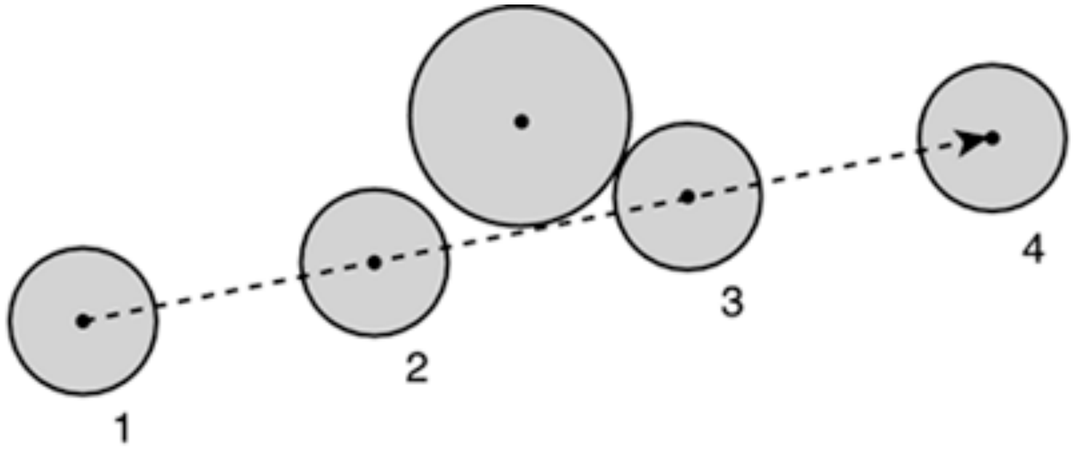
- **The Discrete Time Issue**
  - A typical game updates the state of the game in discrete time slices, such as how you update each object based on the amount of time passed since the last update. For example, in this figure, this bird's-eye view shows an object's discrete time movement. The moving object collides with the larger object in frame 3.



Bird's-eye view of an object's discrete time movement.

Object-to-Object Collisions (14)

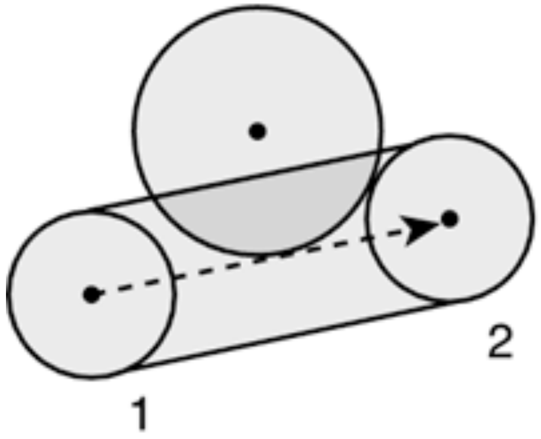
- **The Discrete Time Issue (cont.)**
  - Unfortunately, this discrete time movement can cause a problem with collision detection. Imagine that the object is moving faster or the frame rate is slower. In this scenario, the moving object could "skip over" the object it collides with. For example, in this figure the moving object collides with the larger object between frames 2 and 3.



The problem with discrete time movement: Objects can "skip over" other objects when a collision should be detected.

Object-to-Object Collisions (15)

- **The Discrete Time Issue (cont.)**
  - A couple solutions to this problem exist. The more accurate but computationally expensive solution is to treat a moving object's bounds as a solid shape from the start location to the end location, as shown in this example:



A moving object can be treated as a "tube" to alleviate the discrete time problem.

3. Object-to-World Collisions

Object-to-World Collisions

- Object collisions with the world should generally be as accurate as possible. You don't want the player or another object to move through a wall or to jitter when moving along a wall.

- In Chapter 5, you implemented object-to-world collisions by moving just one coordinate at a time (first x, then y), which worked great for a simple tile-based world in which the objects don't move more than the length of one tile for each frame.

- In the 3D world, you don't have tiles, but you usually have the 3D world described in a structure that can help with collision detection, such as BSP trees. You'll use the 2D BSP tree from the previous chapter to implement collision with floors, ceilings, and walls.

Bounding Boxes for Testing Against Floors

- In the 2D game you used the bounding rectangles of the sprites for collision detections.

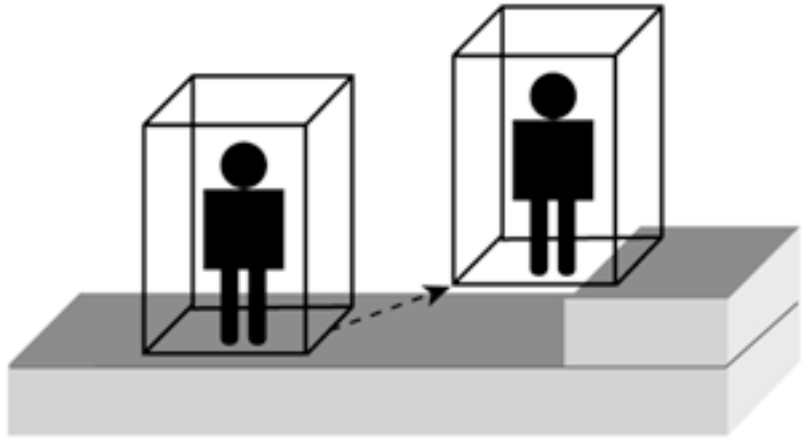
- In 3D, besides bounding spheres, circles, or cylinders, bounding boxes are another popular type of collision-detection mechanism. Two types of bounding boxes exist: **freeform** and **axis-aligned**.

- Freeform bounding boxes can be turned and rotated any way, while axis-aligned boxes are aligned with the x-, y-, and z-axis.

- We use cylinders for object-to-object collision and axis-aligned bounding boxes for object-to-world collisions. The first thing to discuss are floors (and ceilings).

Bounding Boxes for Testing Against Floors (2)

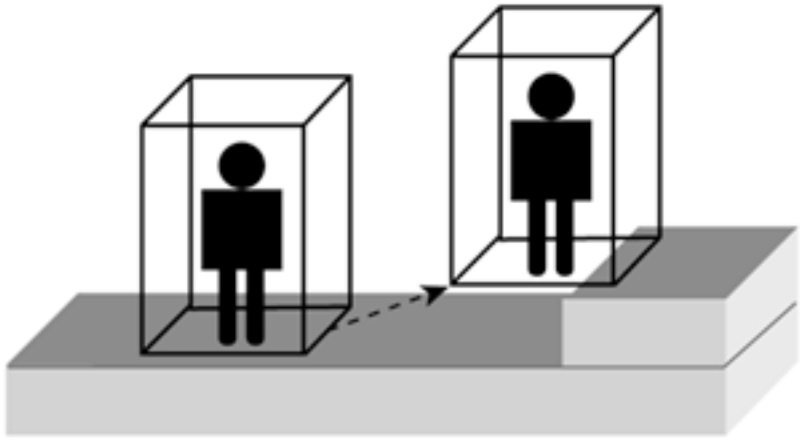
- In a world where floors can be of variable height, you'll want objects to stand on the highest floor under its bounding box, as in the figure.
- Likewise, you don't want objects to move to areas if the ceiling is too low for the object. Also, you might want objects to be able to cross small steps without stopping. In this figure, the player can make the small step up to the platform.



Bounding box collision with a floor: The player's bounding box is partially on the stair, so the height of the stair is used as the player's "floor."

Bounding Boxes for Testing Against Floors (3)

- The highest floor within the object's bounds is used to determine how high the object stands. When testing an object against the floor and ceilings of the environment, you can check each corner of the box for an intersection with the floor or ceiling. This involves four floor checks, one for each corner of the bounding box.



Bounding box collision with a floor: The player's bounding box is partially on the stair, so the height of the stair is used as the player's "floor."

Finding the BSP Leaf for a Specific Location

- With a 2D BSP tree, floor information is stored in the leaves of the tree. You can find the leaf for a specific location pretty easily, similarly to how you traversed the tree.

- This finds the leaf for one location, while an object's bounding box can potentially span multiple leaves. So, you check the leaf of each corner of the bounding box.

```
/**
    Gets the leaf the x,z coordinates are in.
*/
public Leaf getLeaf(float x, float z) {
    return getLeaf(root, x, z);
}


protected Leaf getLeaf(Node node, float x, float z)
{
    if (node == null || node instanceof Leaf) {
        return (Leaf)node;
    }
    int side = node.partition.getSideThin(x, z);
    if (side == BSPLine.BACK) {
        return getLeaf(node.back, x, z);
    }
    else {
        return getLeaf(node.front, x, z);
    }
}
```

Bounding Boxes for Testing Against Walls

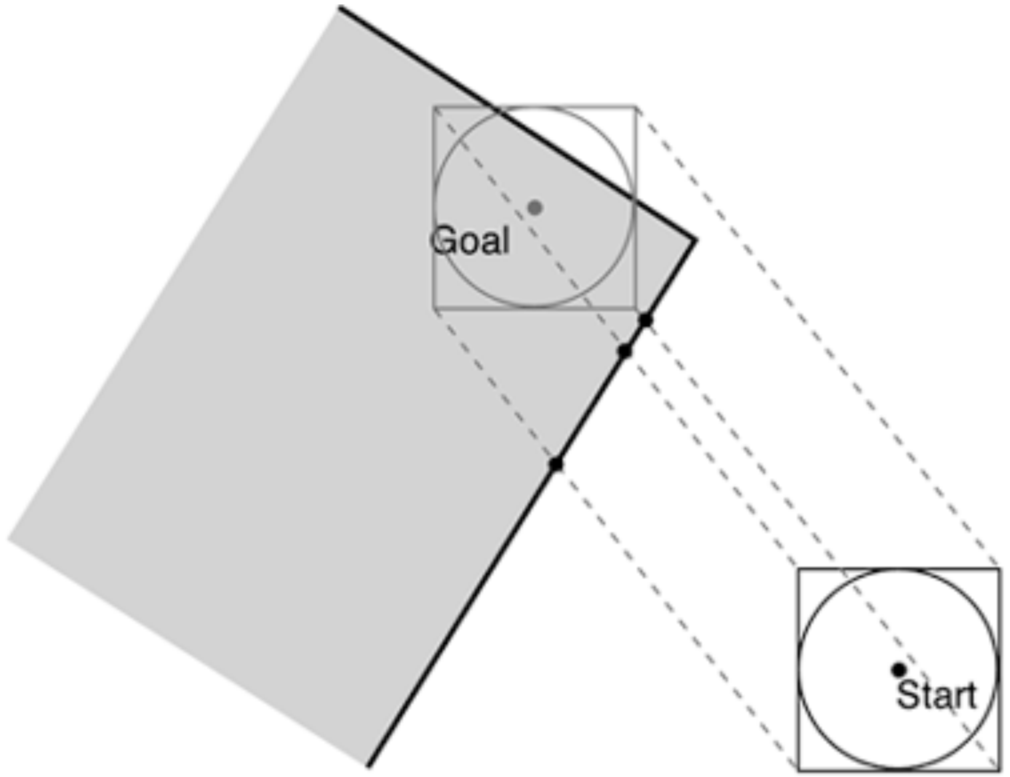
- Walls are thin lines, so if you test for a wall collision only after an object has moved, it's possible you could miss walls that the object has already passed right through. To accurately determine whether an object hits any walls, you need to test the entire path the object travels during the update from one frame to the next.

- For a 2D BSP tree, because an object's path from one frame to the next is a line segment, you can test this line segment for an intersection with any lines that represent walls.

- Objects are solid shapes, not points. So, if you're using bounding boxes for collision testing, you must test all four corners of the bounding box with the walls in a scene.

Bounding Boxes for Testing Against Walls (2)

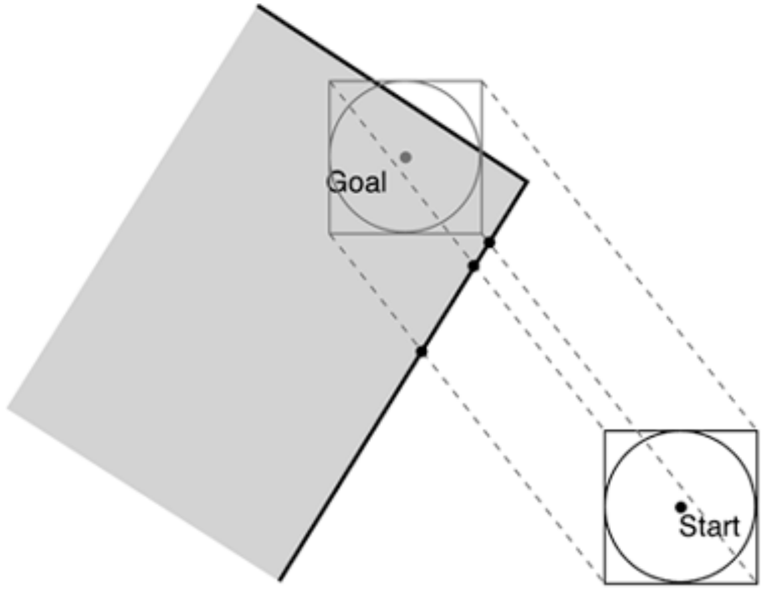
- In this figure, four paths (for each corner) are tested for an intersection, and three of them intersect a wall.

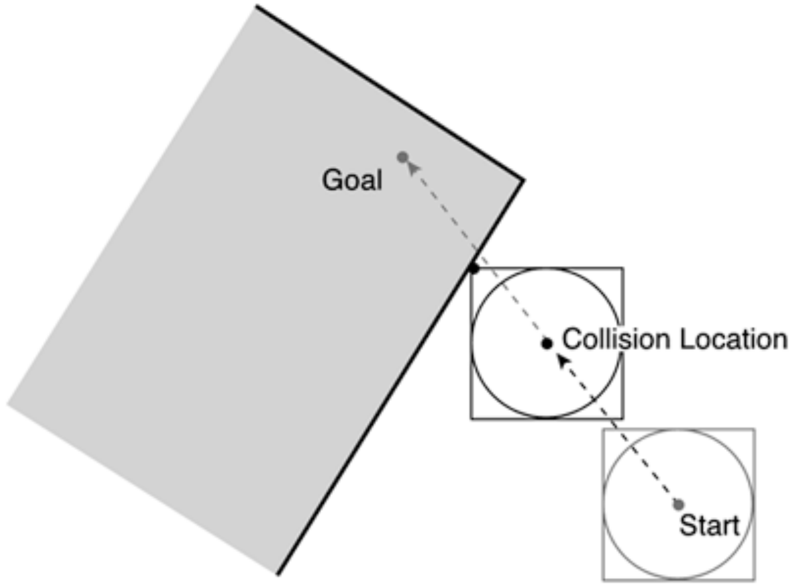


Checking the bounding box corners with an intersection with a wall. Each line segment is checked against the BSP tree for a collision.

Bounding Boxes for Testing Against Walls (3)

- When more than one intersection is found, as in the left figure, the shortest path from the start location to the intersection is the one to use for the collision, as shown in the right figure. Here, the upper-left corner of the object is the first to collide with the wall (it is the shortest path to a line intersection), so it is used to determine the collision location.





Intersection of a Line Segment with a BSP Tree

- Consider a path from (x_1,y_1) to (x_2,y_2). The goal is to find the first intersection of this path with a polygon in the BSP tree, if any. The first intersection is the one closest to (x_1,y_1).
- Here is the algorithm for this, starting with the root node of the BSP tree:
  - Check the path against the node's partition. If the path is either in front of or in back of the node's partition, check the front or back nodes, respectively.
  - Otherwise, if the path spans the node's partition, bisect the path into two paths along the partition. One path represents the first part of the path, and the other path represents the second part of the path.
    - Check the first part of the path for an intersection (see Step 1).
    - If no intersection is found, check the polygons in this node for an intersection.
    - If no intersection is found, check the second part of the path for an intersection (see Step 1).
    - If an intersection is found, return the point of intersection. Otherwise, return null.

Intersection of a Line Segment with a BSP Tree (2)

- This algorithm basically says to look at every partition that the path spans, from the first (closest) partition to the last (farthest) partition, and return the first point of intersection, if any.
- Note that just because a path spans a partition doesn't necessarily mean that an intersection occurred. For an intersection, you need to meet three conditions:
  - For a 2D BSP tree, the line segment representing the polygon spans the path.
  - The polygon isn't a short polygon that the object can step over, and isn't a polygon too high or too low to be in the object's way.
  - The path travels from the front of the polygon to the back of the polygon.
- The algorithm for finding the intersection (and the conditions for an intersection) is implemented in the listing.

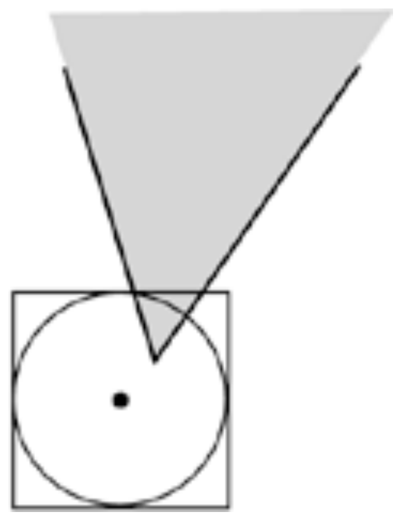
```
Source: CollisionDetection.java
```

Intersection of a Line Segment with a BSP Tree (3)

- In this code, **getFirstWallIntersection()** follows the algorithm mentioned before, and the **getWallCollision()** method checks for the conditions of an intersection between a path and a polygon.
- As a side note, with a 3D BSP tree, you would also use something like this to determine the height of the floor under an object. Just use this algorithm to find the highest polygon underneath the player by checking for the first intersection with a line segment shot straight down from the object.
- You've almost got everything to implement a bounding box collision with a BSP tree, but first there's one issue to look into that could cause problems.

The Corner Issue

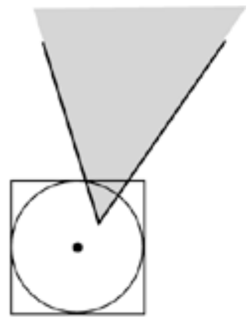
- When you're checking each corner of a bounding box with a collision with the world, you run into conditions in which part of the world will collide with the bounding box but not with the corners of the bounding box, as shown here.



- In this figure, the object collides with a sharp corner that doesn't touch any of the object's bounding box corners.

The Corner Issue (2)

- One way to get around this problem is to treat each edge of the bounding box as a line segment, to test for intersections with the BSP tree.
- If any of the edges intersect with a polygon in the BSP tree, the object is reverted to its original location.
- Alternatively, instead of reverting to its original location, the object could just move back a little bit at a time and test again until no collision is detected.
- Another solution is to ensure that levels are designed so that this issue never arises. However, this puts a restriction on level design that can make things difficult for the level designer because it's very tempting to make corners, as in the example.



Implementing Object-to-World Collision Detection

Checking Walls (CollisionDetection.java)

```
// check walls if x or z position changed
if (object.getX() != oldLocation.x ||
    object.getZ() != oldLocation.z)
{
    checkWalls(object, oldLocation, elapsedTime);
}

...

/**
    Checks for a game object collision with the walls of the
    BSP tree. Returns the first wall collided with, or null if
    there was no collision.
*/
public BSPPolygon checkWalls(GameObject object,
    Vector3D oldLocation, long elapsedTime)
{
    Vector3D v = object.getTransform().getVelocity();
    PolygonGroupBounds bounds = object.getBounds();
    float x = object.getX();
    float y = object.getY();
    float z = object.getZ();
    float r = bounds.getRadius();
    float stepSize = 0;
    if (!object.isFlying()) {
        stepSize = BSPPolygon.PASSABLE_WALL_THRESHOLD;
    }
    float bottom = object.getY() + bounds.getBottomHeight() +
        stepSize;
    float top = object.getY() + bounds.getTopHeight();

    // pick closest intersection of 4 corners
    BSPPolygon closestWall = null;
    float closestDistSq = Float.MAX_VALUE;

    for (int i=0; i<CORNERS.length; i++) {
        float xOffset = r * CORNERS[i].x;
        float zOffset = r * CORNERS[i].y;
        BSPPolygon wall = getFirstWallIntersection(
            oldLocation.x+xOffset, oldLocation.z+zOffset,
            x+xOffset, z+zOffset, bottom, top);

        if (wall != null) {
            float x2 = intersection.x-xOffset;
            float z2 = intersection.y-zOffset;
            float dx = (x2-oldLocation.x);
            float dz = (z2-oldLocation.z);
            float distSq = dx*dx + dz*dz;
```

```
            // pick the wall with the closest distance, or
            // if the distances are equal, pick the current
            // wall if the offset has the same sign as the
            // velocity.
            if (distSq < closestDistSq ||
                (distSq == closestDistSq &&
                MoreMath.sign(xOffset) == MoreMath.sign(v.x) &&
                MoreMath.sign(zOffset) == MoreMath.sign(v.z)))
            {
                closestWall = wall;
                closestDistSq = distSq;
                object.getLocation().setTo(x2, y, z2);
            }
        }
    }
    if (closestWall != null) {
        object.notifyWallCollision();
    }

    // make sure the object bounds is empty
    // (avoid colliding with sharp corners)
    x = object.getX();
    z = object.getZ();
    r-=1;
    for (int i=0; i<CORNERS.length; i++) {
        int next = i+1;
        if (next == CORNERS.length) {
            next = 0;
        }
        // use (r-1) so this doesn't interfere with normal
        // collisions
        float xOffset1 = r * CORNERS[i].x;
        float zOffset1 = r * CORNERS[i].y;
        float xOffset2 = r * CORNERS[next].x;
        float zOffset2 = r * CORNERS[next].y;

        BSPPolygon wall = getFirstWallIntersection(
            x+xOffset1, z+zOffset1, x+xOffset2, z+zOffset2,
            bottom, top);
        if (wall != null) {
            object.notifyWallCollision();
            object.getLocation().setTo(
                oldLocation.x, object.getY(), oldLocation.z);
            return wall;
        }
    }
    return closestWall;
}
```

Implementing Object-to-World Collision Detection (2)

- First, this code tests four paths, one for each corner of the bounding box for an intersection. If more than one intersection is found, the closest one is chosen.
- If two intersections are found an equal distance away, the intersection is chosen that is closest to the direction of the object's velocity. This helps when you actually implement sliding against the wall later in this chapter.
- If an intersection is found, the object's location is set to the collision location, right up next to the wall.
- After those tests, the code tests to ensure that the object bounding box is empty (no corners). If so, the object is reverted to its original location.

4. Basic Collision-Detection Demo

- The **CollisionTest** demo, included with the source code for the book, demonstrates the collision detection so far.

- In this demo, you are stopped when you run against a wall and you can go up stairs to higher platforms. You are also stopped when you run into an object, such as a robot or a crate. The projectiles you fire destroy the robots, and, just for fun, the projectiles stick to the walls, floors, ceilings, and other objects instead of passing through them.

- Collision detection in this demo works great, but it needs some serious help.
  - You get "stuck" when you collide with a wall. In any 3D first- or third-person game, this can be very distracting to the player who expects to "slide" against a wall, especially when moving tightly around corners.

Basic Collision-Detection Demo (2)

  - Movement up and down stairs feels jerky because you are instantly moved up or down instead of scooting smoothly upstairs or allowing gravity to smoothly bring you down.

  - You can't step over small objects in the way. Try shooting a projectile on the floor and stepping over it—you can't.

  - You can't jump.

- Good collision handling is a goal of this chapter, so let's fix those problems! And even though it's not really a collision-handling issue, this will be a good time to implement jumping as well because you can detect when an object is hitting something, even in midair.

5. Collision Handling with Sliding

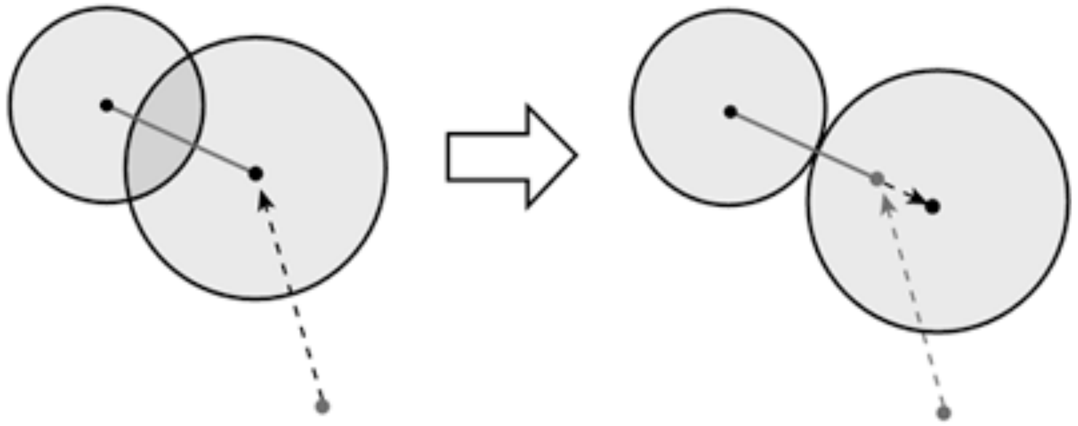
- The next step is to provide collision handling that isn't obtrusive—in other words, the collision handling behaves in a way that the user expects.
- Instead of collision "sticking," you'll implement collision "sliding." For example, instead of stopping when the player hits an object, the player will slide to the side of it. Likewise, the player will slide against a wall and will be able to step over small objects on the floor.
- Also in this section, you'll implement some basic physics to allow objects to smoothly move up stairs, apply gravity, and allow the player to jump.

Object-to-Object Sliding

- Previously, if an object-to-object collision occurred, you just resorted to moving the object to its original location. This is what created the effect of the moving object "sticking" to the static object it collided with.
- To fix this, you must make sure the moving object slides to the side of the static object. The logical solution is to slide the least amount required to get the two objects out of each other's way.
- That means the direction to slide is defined by the vector from the static object's center to the moving object's center.

Object-to-Object Sliding (2)

- In this figure, when the larger object moves, it collides with the smaller object. The large object then moves away from the smaller object so that, after the slide, the two objects are next to one another but not colliding



Object-to-object sliding: The moving object is pushed away from the static object.

Object-to-Object Sliding (3)

- The amount to slide is the difference between the minimum distance and the actual distance:

```
float minDist = objectA.radius + objectB.radius;
float slideDistance = minDist - actualDist;
```

- So, because the vector between the two object's centers has the length of **actualDist**, the formula becomes this:

```
float scale = slideDistance / actualDist;
vector.multiply(scale);
```

- Or, if you know only the square of the distances, it is this:

```
float scale = (float)Math.sqrt(minDistSq / actualDistSq) - 1;
vector.multiply(scale);
```

Object-to-Object Sliding (4)

- Although that square root function is slow, you have to call it only when objects bump into one another, which isn't very often.

- A problem with sliding occurs when sliding against an object actually causes the object to slide into a wall or another object. In this case, the moving object can just revert to its previous location so it still appears to "stick"—the player will have to change direction.

- Object-to-object sliding is easy enough. All the sliding code is kept in the **CollisionDetectionWithSliding** class, which is a subclass of **CollisionDetection**.

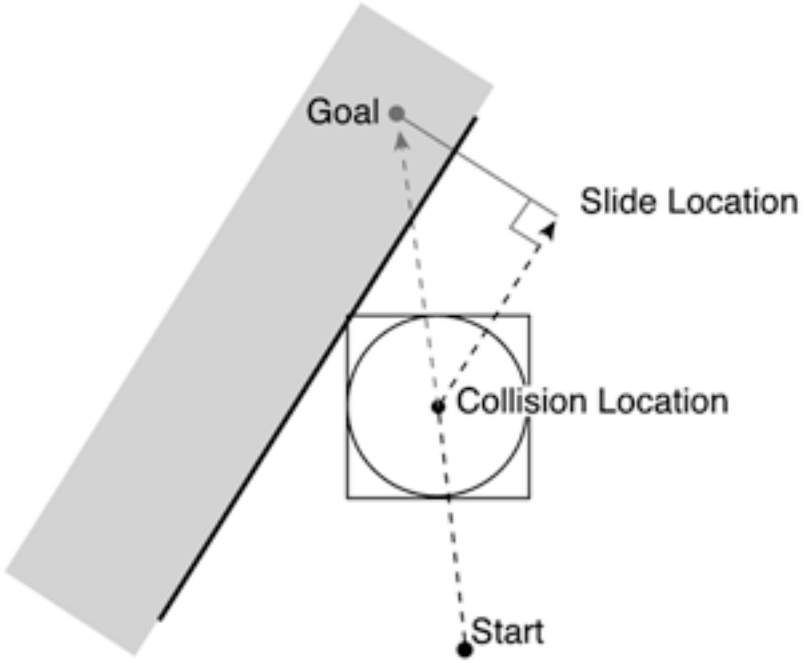
Object-to-Object Sliding (5)

(CollisionDetectionWithSliding.java)

```java
/**
    Handles an object collision. Object A is the moving
    object, and Object B is the object that Object A collided
    with. Object A slides around or steps on top of
    Object B if possible.
*/
protected boolean handleObjectCollision(GameObject objectA,
    GameObject objectB, float distSq, float minDistSq,
    Vector3D oldLocation)
{
    objectA.notifyObjectCollision(objectB);

    if (objectA.isFlying()) {
        return true;
    }

    float stepSize = objectA.getBounds().getTopHeight() / 6;
    Vector3D velocity =
        objectA.getTransform().getVelocity();

    // step up on top of object if possible
    float objectABottom = objectA.getY() +
        objectA.getBounds().getBottomHeight();
    float objectBTop = objectB.getY() +
        objectB.getBounds().getTopHeight();
    if (objectABottom + stepSize > objectBTop &&
        objectBTop +
        objectA.getBounds().getTopHeight() <
        objectA.getCeilHeight())
    {
        objectA.getLocation().y = (objectBTop -
            objectA.getBounds().getBottomHeight());
        if (velocity.y < 0) {
            objectA.setJumping(false);
            // don't let gravity get out of control
            velocity.y = -.01f;
        }
        return false;
    }

    if (objectA.getX() != oldLocation.x ||
        objectA.getZ() != oldLocation.z)
    {
        // slide to the side
        float slideDistFactor =
             (float)Math.sqrt(minDistSq / distSq) - 1;
        scratch.setTo(objectA.getX(), 0, objectA.getZ());
        scratch.subtract(objectB.getX(), 0, objectB.getZ());
        scratch.multiply(slideDistFactor);
        objectA.getLocation().add(scratch);

        // revert location if passing through a wall
        if (super.checkWalls(objectA, oldLocation, 0) != null) {
            return true;
        }

        return false;
    }

    return true;
}
```

Object-to-Wall Sliding

- Sliding along the wall might seem like a complicated task, but it really just involves some simple math. If you know the goal location (the location the object would have moved to had there been no wall), you can easily find the slide location, as shown in this figure.



Object-to-wall sliding: An object slides against a wall.

Object-to-Wall Sliding (2)

- In this figure, the gray line points in the direction of the polygon's normal. You can find the slide location if you can find the length of this line. This is a simple right-triangle problem. Consider the vector to the collision location from the goal location:

```
vector.setTo(actualX, 0, actualZ);
vector.subtract(goalX, 0, goalZ);
```

- Then the length of the gray line is the dot product between this vector and the polygon's normal:

```
float length = vector.getDotProduct(wall.getNormal());
```

- So, the slide location is this:

```
float slideX = goalX + length * wall.getNormal().x;
float slideZ = goalZ + length * wall.getNormal().z;
```

Object-to-Wall Sliding (3)

```java
private Vector3D scratch = new Vector3D();
private Vector3D originalLocation = new Vector3D();

...

/**
    Checks for a game object collision with the walls of the
    BSP tree. Returns the first wall collided with, or null if
    there was no collision. If there is a collision, the
    object slides along the wall and again checks for a
    collision. If a collision occurs on the slide, the object
    reverts back to its old location.
*/
public BSPPolygon checkWalls(GameObject object,
    Vector3D oldLocation, long elapsedTime)
{
    float goalX = object.getX();
    float goalZ = object.getZ();

    BSPPolygon wall = super.checkWalls(object,
        oldLocation, elapsedTime);
    // if collision found and object didn't stop itself
    if (wall != null && object.getTransform().isMoving()) {
        float actualX = object.getX();
        float actualZ = object.getZ();

        // dot product between wall's normal and line to goal
        scratch.setTo(actualX, 0, actualZ);
        scratch.subtract(goalX, 0, goalZ);
        float length = scratch.getDotProduct(wall.getNormal());

        float slideX = goalX + length * wall.getNormal().x;
        float slideZ = goalZ + length * wall.getNormal().z;
```

```java
        object.getLocation().setTo(
            slideX, object.getY(), slideZ);
        originalLocation.setTo(oldLocation);
        oldLocation.setTo(actualX, oldLocation.y, actualZ);

        // use a smaller radius for sliding
        PolygonGroupBounds bounds = object.getBounds();
        float originalRadius = bounds.getRadius();
        bounds.setRadius(originalRadius-1);

        // check for collision with slide position
        BSPPolygon wall2 = super.checkWalls(object,
            oldLocation, elapsedTime);

        // restore changed parameters
        oldLocation.setTo(originalLocation);
        bounds.setRadius(originalRadius);

        if (wall2 != null) {
            object.getLocation().setTo(
                actualX, object.getY(), actualZ);
            return wall2;
        }
    }

    return wall;
}
```

Object-to-Wall Sliding (4)

- In this code, the **checkWalls()** method of CollisionDetection is overridden. First the slide is applied; then it checks whether any wall collisions occurred after the slide. If so, the object is moved back to the collision location.

- You can always disable object-to-wall sliding. Projectiles, for example, override the notify methods so that projectiles stop when they hit a wall, floor, or ceiling:

```
public void notifyWallCollision() {
    transform.getVelocity().setTo(0,0,0);
}


public void notifyFloorCollision() {
    transform.getVelocity().setTo(0,0,0);
}


public void notifyCeilingCollision() {
    transform.getVelocity().setTo(0,0,0);
}
```

Object-to-Wall Sliding (5)

- This creates the effect of a projectile "sticking" to a wall or other object, so you can emulate the previous collision-handling technique.
- Now you've got sliding for objects and walls. Next up: sliding up stairs (and a little gravity).

Gravity and Sliding Up Stairs (Object-to-Floor Sliding)

- Another common sliding effect is to allow the player and other objects to slide smoothly up stairs. Otherwise, as in the first collision-detection demo, the player seems to jitter when moving up stairs because the y location of the object instantly changes to the higher stair step.

- Also, when you move from a higher platform to a lower one, you instantly drop to the lower level instead of gradually dropping because of gravity.

- Gravity will work the same as in Chapter 5, by accelerating the object's downward velocity as time progresses. If the object's y location is higher than the floor's y location, gravity can be applied to the object.

Gravity and Sliding Up Stairs (Object-to-Floor Sliding) (2)

- The opposite is true for sliding up stairs: If the object's y location is lower than the floor's y location, a scoot-up acceleration can be applied to the object.

- The physics used for the game is summed up in the Physics class. This is just a start, and you'll add more to it later.

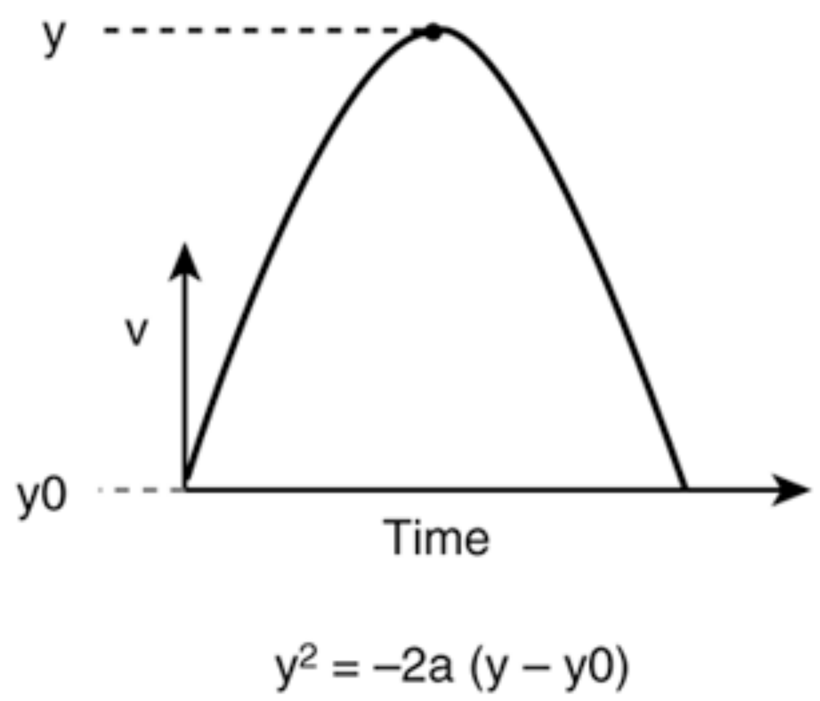
```
/**
    Default gravity in units per millisecond squared
*/
public static final float DEFAULT_GRAVITY_ACCEL = -.002f;

/**
    Default scoot-up (acceleration traveling up stairs)
    in units per millisecond squared.
*/
public static final float DEFAULT_SCOOT_ACCEL = .006f;

private float gravityAccel = DEFAULT_GRAVITY_ACCEL;
private float scootAccel = DEFAULT_SCOOT_ACCEL;
private Vector3D velocity = new Vector3D();

/**
    Applies gravity to the specified GameObject according
    to the amount of time that has passed.
*/
public void applyGravity(GameObject object, long elapsedTime) {
    velocity.setTo(0, gravityAccel * elapsedTime, 0);
    object.getTransform().addVelocity(velocity);
}


/**
    Applies the scoot-up acceleration to the specified
    GameObject according to the amount of time that has passed.
*/
public void scootUp(GameObject object, long elapsedTime) {
    velocity.setTo(0, scootAccel * elapsedTime, 0);
    object.getTransform().addVelocity(velocity);
}
```

Gravity and Scooting (Physics.java)

Make It Jump

- Jumping works pretty much the same as it did in Chapter 5—just apply an upward velocity to an object. The power of this initial velocity diminishes as gravity is applied.

- Sometimes, though, you want to guarantee how high an object can jump. The figure shows the equation.



Make It Jump (2)

- Add a few more methods to the Physics class to implement jumping.
- These methods enable you to find the jump velocity needed to jump a certain height (based on the gravity acceleration) and to actually make an object jump.

```
/**
    Sets the specified GameObject's vertical velocity to jump
    to the specified height. Calls getJumpVelocity() to
    calculate the velocity, which uses the Math.sqrt()
    function.
*/
public void jumpToHeight(GameObject object, float jumpHeight)
{
    jump(object, getJumpVelocity(jumpHeight));
}


/**
    Sets the specified GameObject's vertical velocity to the
    specified jump velocity.
*/
public void jump(GameObject object, float jumpVelocity) {
    velocity.setTo(0, jumpVelocity, 0);
    object.getTransform().getVelocity().y = 0;
    object.getTransform().addVelocity(velocity);
}


/**
    Returns the vertical velocity needed to jump the specified
    height (based on current gravity). Uses the Math.sqrt()
    function.
*/
public float getJumpVelocity(float jumpHeight) {
    // use velocity/acceleration formal: v*v = -2 * a(y-y0)
    // (v is jump velocity, a is accel, y-y0 is max height)
    return (float)Math.sqrt(-2*gravityAccel*jumpHeight);
}
```

6. Collision Detection with Sliding Demo

- That's it for collision detection. Included with the source for this chapter is the **CollisionTestWithSliding** class, which is exactly the same as the previous demo except that it implements sliding. All in all, it demonstrates sliding against a wall, sliding against objects, stepping over objects, getting on top of objects, scooting up stairs, applying gravity, and jumping.

- As a side effect to the engine, you can even stand on top of projectiles. Because projectiles stick to walls, you can fire enough projectiles to try to "draw" a ramp of them against the wall. Then you can run up the ramp!

```
Source: CollisionTestWithSliding.java
```

7 . Enhancements

- You got some great basic collision detection and handling in this chapter, but there's always room for improvement. Here are a few ideas:
  - Implement sphere trees for more accurate object-to-object tests.
  - Perform extra checks for whether an object travels a large enough distance in between frames; a collision could occur between the start and end locations.
  - Allow the player to crouch or crawl to get in tight places.
  - Physics-wise, you could implement "head bobbing," which just bounces the camera up and down when the player is moving.

8 . Summary

- Collision detection: one of the necessities of almost any game, 2D or 3D.
- First we talked about isolating objects on a grid to limit the number of actual collision tests to make. Then we talked about various collision-detection mechanisms, such as *bounding spheres*, *sphere trees*, *bounding cylinders*, and *bounding boxes*.
- We implemented collision detection with other objects and with the walls, floors, and ceilings of a BSP tree. We implemented some better *collision handling* that enables the player (and other objects) to slide against other objects' walls, slide against walls, and scoot up stairs. Finally, we added *gravity* and *jumping* for good measure.
- You can blast the robots to smithereens, but they aren't much of a challenge, are they? That's what you'll work on next: giving your enemies some artificial intelligence so you can make the games you create more fun and challenging.