## SATURDAY, AUGUST 1, 2009

### [Detecting and responding to dynamic collisions.](http://replicaisland.blogspot.de/2009/08/detecting-and-responding-to-dynamic.html)

In the last post I discussed how Replica Island uses line segments organized as a 2D regular grid of tiles as the basis for its background collision system. This time I will explain how dynamic collisions (collisions between moving objects) are detected and resolved.  
  
In Replica Island I draw a distinction between collisions that occur with the background geometry (falling on the ground, sliding on a slope, hitting the Android's head on the ceiling) and collisions that occur between game objects (the Android depressing a button, or hitting an enemy, or collecting a coin). While both of those cases are forms of "collision detection," they represent very different types of tests and I have two (entirely separate--mostly) systems for dealing with them.

#### Moving objects vs the background

Since I alluded to how background collision detection works in the last post, I'll start with that system. If Android is falling through space and passes into the ground, I need to detect that intersection and then fix it so that he doesn't actually fall through the floor. This is actually a pretty tricky problem because the frame rate of any Android game (and really, any modern game on any platform) can fluctuate as the game is played. The Android moves through space in game units / second, but the game is displayed in frames per second, and depending on the current speed of the game there's no good way to predict how far he'll move in a single frame. So I need a method that can cover a range of space between the last frame and this one so that even a dramatic movement won't allow the player to pass through walls.  
  
The solution is to "sweep" the space in between the character's position at the last frame and his current position, and snap the character back if an intersection is detected. I use rays to do this: rays are cast from a character's previous position to his current position, and the first intersection along the ray (that is, the intersection that is closest to the ray's start point) is considered to be the spot at which the character hit a wall. I also filter my ray test by the normals of the surfaces I am considering; surfaces that do not oppose the direction of the ray can be ignored (that is, I only care about surfaces whose dot product against the direction of the ray is less that 0). Characters are not well described by a single, thin ray, however, so I do two tests: one filtered against horizontal surfaces and one filtered against vertical surfaces (angled surfaces fall into one bucket or the other depending on their slope). This is a nice method because it allows me to tune exactly how I test a volume against collision; often I want to allow a small amount of intersection: when the character is standing on a sloped surface, for example, I want to allow his bounding box to intersect with the slope slightly so that it looks like his feet are on the ground. With a couple of simple ray tests this method covers the space between the previous frame and the current frame pretty well without too much processor overhead. I briefly experimented with a final volume test pass to make sure that collisions were not being missed by the ray tests, but in the end such a test wasn't necessary (and actually, despite being more technically correct, the results were a lot less fun).  
  
Sometimes I want game objects that act like background collision but are not part of the collision tile map. For example, a moving platform might act in every way like a background element except that it can move. In these cases, I allow entities to submit temporary line segments to the background collision system which will then be used along with all the rest of the background collision line segment data. This way characters in the world can be made to act exactly like solid objects without remaining static, and other characters can come along and react to them without any special code.

#### Moving objects vs each other

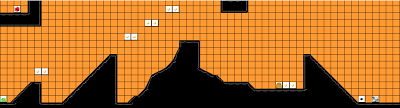
However, the more common case is when two game objects--a bullet and the player, an enemy and some spikes, etc--that are moving and non-solid come into contact. In this case we need to detect the intersection and then let code specific to each entity decide what to do about it. Still, we can generalize the system a little bit more: usually in such collisions we can name one of the entities as the "offender" and the other entity as the "victim." When a bullet hits the player, the bullet is the offender and the player is the victim. When a robot runs into some spikes, the spikes are the offender and the robot the victim. In fact, if we consider an animating character, we might want to mark some parts of a given frame of animation as "offensive" and other parts "vulnerable." In a game where a character can punch, we probably want the character's fist to deal damage to other characters but at the same time we'd expect other parts of the character, say his back and head, to be vulnerable to hits. So in order to detect collisions between game entities, I decided to give my entities multiple collision volumes, some associated with offensive areas and others associated with areas that are vulnerable to hits.  
  
Each animation frame in Replica Island can carry a list of "attack" volumes and "vulnerability" volumes. When detecting collisions, I stipulate that collisions can only occur between a single attack volume and a single vulnerability volume. Furthermore, volumes can be set to deal and receive specific types of hits, which allows me to filter the number of actual volume intersection tests I need to perform (for example, the coin is only vulnerable to a "collection" hit, so only collection-hit-dealing attack volumes will be tested against the coin.  
  
Each time an animation frame changes a new set of attack and vulnerability volumes may become active. These volumes are unioned together into a sphere that is guaranteed to encompass them called the *bounding sphere*. The volumes, along with the bounding sphere, are then submitted to the runtime collision detection system. Each frame, the collision detection system sorts all of the bounding spheres that have been submitted and tests them for intersections. The sort is by the left-most point of the sphere, so objects end up sorted along the x-axis of the level. This is a type of [sweep and prune algorithm](http://en.wikipedia.org/wiki/Sweep_and_prune), and it makes it easy to quickly find overlapping bounding spheres because potentially colliding sphere pairs are guaranteed to be grouped together in the sorted list. When a pair of bounding spheres that intersect is found, each of the related entities' attack and vulnerability volumes are tested for intersection. If an intersection between an attack volume and a vulnerability volume is found, we know that these two entities have hit each other and some other code needs to run in order to respond.  
  
For a long time the Replica Island engine only supported sphere collision volumes for these kinds of dynamic tests. About half-way through development I added an axis-aligned box collision type as well, but otherwise no complicated collision volume tests have been necessary. I'm very happy with the way that this system turned out: it's reliable, fast, and easy to extend.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224" \o "author profile)AT [7:13 AM](http://replicaisland.blogspot.de/2009/08/detecting-and-responding-to-dynamic.html" \o "permanent link) [4 COMMENTS:](http://replicaisland.blogspot.de/2009/08/detecting-and-responding-to-dynamic.html" \l "comment-form)

LABELS: [CODE DESIGN](http://replicaisland.blogspot.de/search/label/code%20design)

### [Fast and accurate 2D collision detection with line segments.](http://replicaisland.blogspot.de/2009/08/fast-and-accurate-2d-collision.html)

Replica Island is a tile-based game, which means that the levels are laid out using small (32x32), reusable tiles. I chose this approach for two reasons: it's memory efficient and exceedingly common for this genre. With a tile-based game you define a set of tiles--in the case of Replica Island, a single texture representing each possible tile--and then draw the level by combining those tiles together. Even if you have a lot of levels you don't need to spend a lot of disk space (or runtime memory) on the actual art for the tiles, and the data describing the layout of each level is very small. So tiles are useful for games that have a lot of levels and wish keep the total size of their application down. Tiles are also the standard way to make side scrolling games. Pre-Sony Playstation 1 game hardware (and some post-PS1 hardware, like the Nintendo GameBoy Advance) was hardwired to deal with tiles because of these efficient properties, and as a result the vast majority of side-scrolling games are tile-based. So going with a tile-based game engine also made sense because it helps the game feel like a proper side scroller.  
  
  
So when it came time to write a collision system to represent level geometry for Replica Island, tiles seemed like a natural choice. I already had a tool for editing tiles and tile maps, as well as runtime code for loading tile sets and maps, so using tiles for collision meant I could leverage a lot of existing infrastructure. Also, maintaining a 2D array of collision tiles in memory was appealing because it's very fast to query the contents of any particular tile; while many collision systems organize their data in trees, doing so requires that the tree be traversed for every collision test. A 2D array, on the other hand, can be indexed directly into, which is very fast (the down side is that these arrays are generally sparse, so there is a lot of runtime memory wasted; however, for the size of the levels I was considering and the available ram on Android devices, the memory required of an array to describe the level is trivially small).

[](http://3.bp.blogspot.com/_EIf-QLT1Gfk/SnRSq_I9oGI/AAAAAAAAABU/bCi3FY_PfaM/s1600-h/collision_level.png)  
*The collision tile layout for an early test level.*

#### Tile-Based Line Segment Collision? Huh?

The problem with using tiles for collision is that they are basically square. The simplest implementation is to simply consider the collision world a 2D grid of booleans, set to true in grid cells that are "solid" and false to cells that are "empty." Using this method you can calcuate the location of a collision cell in game space and then check to see if it is solid or not before deciding if, for example, your player can move forward. While that's a very simple approach (and I shipped a few games that worked that way back in the day), it's not really going to scale to the types of interesting level designs that modern players expect. At the very least you want to be able to support a sloped surface so you can make hills and valleys, and ideally you should be able to express bumps, spikes, curves, and other shapes that are much more complicated than a simple solid cube.  
  
So, for Replica Island, I decided to implement a collision system based on arbitrary line segments, stored as shapes in tiles and laid out in the world as a tile map. The idea is pretty straightforward: within a 32x32 collision tile I can define any number of line segments, which can have arbitrary angles and normals. I can use my tile editor tools to lay the collision tiles out and at runtime I can leverage the speed of a 2D array (direct index into the current tile) to find a set of potentially-intersecting line segments. The nice thing about line segments is that they can contain both a slope and a normal, which allows for all kinds of interesting physics calculations without a lot of code (for example, to make a ball bounce convincingly off an angled surface you can simply reflect the ball's velocity about the normal of the surface). I've used similar systems (though never stored as tiles) on a lot of other games and, at least for 2D collision worlds, I'm quite happy with the approach.

#### Data Generation

Once I had settled on the line-segments-in-tiles approach, the problem became actually generating the line segment data for each tile. Genki, the artist behind Replica Island, generated a set of collision tiles that would serve as the basic building blocks of our levels, but now I needed a way to represent that same data as line segments. I've had success with edge-tracing algorithms in the past but my experience with them also suggested that they require a bit of tuning to get right. And since collision detection and response is so closely tied into core game play, I wanted a way to hand-modify individual segments. When I worked in games development full time we would just sit down and write an editor tool for this kind of thing, but one of my goals with Replica Island is to see how simply (read: cheaply) I can get it finished, so writing a dedicated tool was a little out of scope. So I decided to piggyback on some existing tool to generate the data that I needed. After thinking about it for a bit, I settled on Photoshop.

[](http://2.bp.blogspot.com/_EIf-QLT1Gfk/SnRR0fhO5sI/AAAAAAAAABM/5XTDjB3teuc/s1600-h/collision.png)  
*The collision tileset used to build levels in Replica Island.*

Did you know that you can actually script Photoshop with Javascript? I did not know this, but it's actually pretty easy to do (if neigh undebuggable). After a day of futzing around with Photoshop's Javascript API I had a tool that would walk Photoshop paths and generate a list of line segments with normals (calculated by requiring the path be closed and by assuming that all normals point away from the centroid of the path shape). Since I couldn't figure out a way to actually write out a text file from Photoshop, the script opens a new document, creates a text layer, and then dumps its output into that layer. Once the script was written I went over Genki's collision tileset with the path tool and generated unique paths for each tile. I also added a very simple tool to take the text output of my script and pack it as binary data for loading at runtime. It took a total of about three days to go from having no collision data to having a full tileset of data, ready for use at runtime.

#### Querying Collision at Runtime

So, now I have two different sets of data: I have a single collision tile set, which maps collision tile indicies to collections of line segments, and a bunch of collision tile maps (one for each level) that place individual collision tiles in space to describe level geometry. I can load the collision tile set once and keep it around; each level that I load brings with it its own collision tile map. At runtime, to check to see if a region of space in the game world is blocked or not, I can cast a ray through the tile map (using one of my favorite algorithms: the [Bresenham line algorithm](http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)) and check it against the collision tiles that it touches. The actual line segment vs line segment test is pretty simple (here's a [useful reference](http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline2d/)), and it produces an exact intersection point. As soon as an intersection is detected the ray test can be aborted and the results returned (note that it's necessary to test the ray against all segments in a particular tile, though; you always want the intersection point closest to the origin of the ray, so all segments within a given tile must be tested and only the closest intersection returned). It's also possible to test a collision volume against the world by simply visiting each cell that intersects the volume and doing a volume vs line test, but in Replica Island this sort of test did not end up being necessary. The game play is described using ray casts alone.  
  
So, given a collection of collision tiles, each containing a collision of line segments, and a map of those tiles laid out on a 2D grid, I was able to make a pretty expressive 2D collision system without a whole lot of effort. The next tricky bit, which I'll cover in a subsequent post, was what sort of tests, and what sort of response, are actually the best for game play. And this system only deals with intersections between entities and the background; in the future I'll write about the system I used to detect collisions between individual entities.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [6:45 AM](http://replicaisland.blogspot.de/2009/08/fast-and-accurate-2d-collision.html) [4 COMMENTS:](http://replicaisland.blogspot.de/2009/08/fast-and-accurate-2d-collision.html#comment-form)

LABELS: [CODE DESIGN](http://replicaisland.blogspot.de/search/label/code%20design)

### [Replica Island Development Blog!](http://replicaisland.blogspot.de/2009/08/replica-island-development-blog.html)

This is the first post for this new-fangled Replica Island development blog. I will use this space to talk about the development of Replica Island, both in terms of code and game design. Since this is the first post, here's the low-down on the game:

* Replica Island is a 2D side-scrolling platformer for Android devices.
* It stars the Android robot, who (it turns out) has a pretty devastating butt stomp move.
* The source for Replica Island will be released with the game. It's written in Java.
* Replica Island is currently in Alpha. I'm working hard on finishing it up.
* When it is released, Replica Island will be available for free on Android Market.
* The official Replica Island web site is [http://www.replicaisland.net.](http://www.replicaisland.net/)