## THURSDAY, SEPTEMBER 10, 2009

### [Why Android Flies](http://replicaisland.blogspot.de/2009/09/why-android-flies.html)

The main mode of movement in Replica Island is flying. The Android robot has thrusters in his feet, and while he can also slide upon the ground, most of the game is spent maneuvering through the air. Enemies are dispatched by dropping on them from above and the collision and physics systems make it easy to glide off of a ramp and into the air. I don't think this is a common approach for side scrolling platformers.  
  
The reason that the Android flies is that many Android devices ship with a trackball. Some (like the Samsung Galaxy) have a d-pad, which works fine as well, but the devices from HTC (which include the G1, myTouch, and Hero) all sport a trackball. The great thing about a trackball is that it's an analog interface with a pretty high resolution. The bad thing about it is that you can't hold a single direction down like you can with a d-pad; to keep sending input events in a specific direction, you have to keep rolling in that direction.   
  
When I started working on Replica Island, I knew that I'd have to do something different with the player controls to afford trackball use. Letting the Android robot fly seemed like a good solution: the player gets going in a certain direction and then only has to make minor adjustments in heading to steer rather than constantly depressing a button indicating the direction that they wish to travel. This sort control scheme is a bit like a car steering wheel--given velocity that you already have you just need to worry about turning left and right. Indeed, in early prototypes it was clear that the trackball would work very well for this sort of analog maneuvering, but I also quickly learned that it's hard to make precise movements with a trackball.   
  
My initial implementation let the player move left, right, and up by rolling the ball, and while it worked and was playable, people had a really hard time moving in only one direction. It was too easy to accidentally roll up a little when trying to go left or right, which made the controls feel unwieldy and arbitrary. After experimenting with dead zones and other input filters, I finally reduced the trackball to horizontal movement and put a jump/fly button at the lower-left hand corner of the screen. This is a much better solution because it allows players to manage their horizontal and vertical velocity independently, much like they can with a d-pad or other traditional game input device. Furthermore, by ignoring vertical movement on the trackball the control scheme became a lot more forgiving to minor input mistakes.  
  
Replica Island is playable with three inputs: the jump/fly button, left and right controls, and a drop/attack button. These three inputs were decided upon to allow the trackball to feel good, but they map well to other input systems (like a hardware keyboard or d-pad) too, so Replica Island is playable on a wide range of hardware.  
  
Once I settled on the trackball-based steering approach for movement, many other aspects of the core game design fell out. I posted earlier about the collision detection system I wrote to support slopes and angles; this system was necessary to allow for ramp and bounce physics. The Android robot can slide across the ground but it's a lot more fun when he's in the air, so I needed to have a collision and physics system that allows him to take flight easily. Sliding down hills or going off ramps feels good and works well with the control scheme.  
  
This approach also simplified animation: if the Android robot is going to fly with thrusters in his feet, he doesn't need a walk cycle, a run cycle, a turn animation, a skid to stop animation, a jump windup or a jump peak or a jump fall animation. His motion system is more complicated than most of the side scrollers I've made in the past but his animation system is extremely simple; the control scheme afforded this simplicity.  
  
As awesome as Android is for games, the devices it runs on are not generally designed explicitly for gaming. To make a fun game, especially one that is part of an established console-oriented genre, you need to come up with a control scheme that works well for the hardware, even if there's not a lot of precedent for it. In the case of Replica Island, I started with the control scheme and the rest of the game design flowed out from there; as a result, I think that it's a quite playable platformer despite its rather unconventional controls.   
  
When making games for this type of system, it's important to consider exactly how the available inputs are going to make the game fun; if you try to just glue controls from some other hardware paradigm onto a game running on a phone the results are almost guaranteed to suck. Getting the controls right is probably the single most important task you have as a game developer, and if you can nail them down early enough, a huge part of your game design will pretty much write itself.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [6:32 AM](http://replicaisland.blogspot.de/2009/09/why-android-flies.html) [5 COMMENTS:](http://replicaisland.blogspot.de/2009/09/why-android-flies.html#comment-form)

LABELS: [GAME DESIGN](http://replicaisland.blogspot.de/search/label/game%20design), [TARGETING ANDROID](http://replicaisland.blogspot.de/search/label/targeting%20Android)

## TUESDAY, SEPTEMBER 8, 2009

### [Aggregate Objects via Components](http://replicaisland.blogspot.de/2009/09/aggregate-objects-via-components.html)

I used to make games like this:

class RenderableMovingCollidableGameObject extends RenderableMovingGameObject {

public void update() {

super.update(); // Parent classes implement rendering and movement.

// implement collision detection here

}

}

class PlayerObject extends RenderableMovingCollidableGameObject {

public void update() {

super.update(); // Run rendering, movement, and collision.

// update the player

}

}

...

// Main loop!

while (true) {

InputSystem.update(); // poll for input.

for (gameObject : ListOGameObjects) {

gameObject.update();

gameObject.draw();

}

}

This isn't a bad way to start making games, but it doesn't scale. Making a good game requires flexibility and the ability to iterate quickly, and this approach starts to break down when the game becomes medium-sized. What if we have more than one game mode? What if the behavior of a specific game object needs to change for a short period of time? What if Level 27's version of the player needs to be subtly different than Level 26's?  
  
In a nutshell, the problem with this approach is that the structure of the code producing a single frame of the game is hard-coded into the program. What I want is something that can change its structure between levels, or between game modes, or even on the fly. I need something more dynamic than a hard-coded for loop.  
  
**Another Approach**  
  
Let's look at game objects as an example. Game objects are entities in the game world like coins, enemies, moving platforms, and the player. They often have similar functionality, so one way to go about implementing a set of similar objects is to use an inheritance tree. We can create a base GameObject class, and from that derive RenderableGameObject, and from that derive RenderableMovingGameObject, and from that derive RenderableMovingCollidableGameObject, etc, etc, etc. Each level of derivation can add some common functionality until the leafs of this tree are specific entities, like the player.  
  
The real problem with inheritance trees is that features that don't need to be inter-related become dependent on each other because of the way the code itself is written. Given the example class structure above, it's not possible to make a GameObject that can move but doesn't need to render (short of littering the code with flags--don't do that). Because of inheritance, a dependency between movement and rendering has been created where none actually needs to exist. What if we could mix and match features on a per-instance basis rather than tying them all together in a class hierarchy? This is where object composition comes in.  
  
Object composition (or "object aggregation" depending on which design patterns book you read) is the idea that an object "has a" feature instead of "is a" feature. Rather than using inheritance or some other code-side method of collecting functionality together in a single object, we make the object manage a list of separate feature objects; the contents of that list can be different per instance.   
  
So for Replica Island, I have a GameObject class that contains a list of GameComponents. A game object with an empty list does nothing; it can't be drawn, or make noise, or hit things, or do anything else to affect the game. GameComponents implement all of those features, and they must be inserted into the GameObject for it to actually be able to act. Here's some psudeo-code of how GameObject and GameComponents work:

class GameObject {

private Array<GameComponent> mComponents;

public void addComponent(GameComponent component) {

mComponents.push(component);

}

public void update(float time) {

for (component : mComponents) {

component.update(time, this);

}

}

}

class GameComponent {

public void update(float time, GameObject parent) {

// ... functionality goes here

}

}

A GameObject just runs all the GameComponents it contains to produce its output each frame. GameComponents in Replica Island implement things like movement, physics, collision detection, sprite animation, rendering, animation selection, AI, player control, etc. Once a generic GameComponent is written it can be inserted in any number of objects; if a specific object requires special functionality, a new GameComponent can be written just for that object without disturbing anything else in the system.  
  
The beauty of this approach is that individual components can be compartmentalized features, and brand new objects in the game world can be created just by sticking different pieces of pre-existing code together in new ways. The result is also dynamic: unlike an inheritance-based object, Replica Island game objects can change their structure at runtime. For example, one of the Android robot's powers is to possess other robots. When an enemy robot is possessed, his AI component is removed and a new component that lets the player control him is inserted. All of a sudden the robot is being driven by the player; all the rest of his code for animation selection, physics, and collision detection, continues to work without realizing that anything has changed. When the player releases the robot, the original component structure can be restored (actually, in this case the robot blows up, taking out anything else near it, but you get the idea).  
  
I've made a couple of games now using components and I'm very happy with the result. In Replica Island I took a couple of shortcuts for speed that damage the goal of completely independent components, but I think it was worth it; the sacrifices I've made for frame rate haven't actually proved detrimental to the flexibility or extensibility of the system.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [7:55 PM](http://replicaisland.blogspot.de/2009/09/aggregate-objects-via-components.html) [21 COMMENTS:](http://replicaisland.blogspot.de/2009/09/aggregate-objects-via-components.html#comment-form)

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