Extracted from:

Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Google Glass

Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

Edited by Jacquelyn Carter

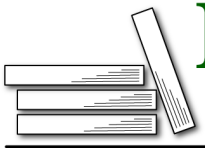
Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-18-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—October 22, 2014

Video games can be excellent tests of a computer system's abilities. They can push the limits of CPU/GPU, memory, display, audio, inputs, and in some cases of networked games, bandwidth. In the previous chapter, we dipped our toes into Glass immersions. But now it's time to dive in, headfirst, and see just how deep an immersive experience can go by creating a Google Glass video game. We already have many of the tools necessary to make any form of Glassware, so we'll create a side-scrolling adventure game called Miss Pitts. This is a jumping-style game inspired by the endlessly annoying the Flappy Bird genre.

In this chapter we're constructing this game, designed as an immersion activity running outside the timeline, with a simple splash screen that shows the game's high score. From there we'll design how the user interacts with the game by using the gesture bar and 9-axis sensor inputs, followed by rendering graphics, sound effects and music. Finally, we'll fill in the rest of the game with the objects and game engine logic required to make the whole thing go.

A Note On Code in this Chapter

There's a lot of code in this chapter. Throughout this book we've forgone printing certain lines of Java code in the interest of saving space. For example, we always skip `package` and `import` statements. This game constitutes even more code, far too much to print out. So we'll be skipping most obvious class field definitions, trivial constructors, and required `try` statements with empty `catches`. More than with any chapter before, I'd suggest you download the code and follow along.

Start The Game with a Splash Screen

The start of any good video game design is explaining to your users why they're here, and what they're trying to achieve. A backstory, no matter how basic, is a good launch point. Here's our story:

Our heroine, Miss Pitts, lives in a post-apocalyptic two dimensional world, where huge sinkholes have appeared in the only street that actually exists. If she falls down a pit it's game over. Luckily, she can run to the right, jumping at varying heights, allowing her to clear differently sized pits. Let's keep the scoring and game play simple: You receive one point per pit jumped.

It's a good idea to sketch out our game before we start coding. Happily our game is pretty easy, and can be described in a single screen, like so:

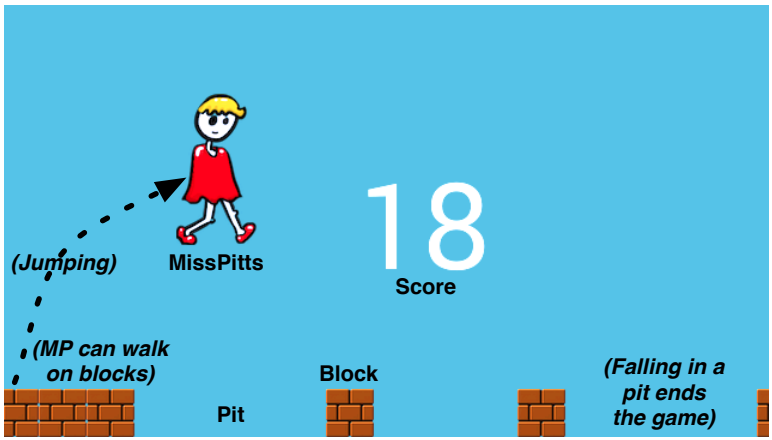


Figure 48—Designing MissPitts in One Screen

Since this is a standard immersion, we need to start with an activity which we'll just give the deeply clever name GameActivity. Don't forget to set android:immersive="true" in the Android manifest.

Voice Commands

There's one minor deviation from how we've created voice triggers so far in this book. In MissPitts, We're going to use the built-in voice command PLAY_A_GAME. In our voice_trigger.xml, rather than giving the trigger a keyword, we're instead giving it the built-in command.

chapter-14/MissPitts/res/xml/voice_trigger.xml

```
<?xml version="1.0" encoding="utf-8"?>
<trigger command="PLAY_A_GAME" />
```

You can get the entire list of commands from the VoiceTriggers.Command class.[1] Here are a few examples:

ADD_AN_EVENT	CHECK_THIS_OUT	EXPLORE_THE_STARS
CALCULATE	CONTROL_MY_CAR	FIND_A_BIKE
CALL_ME_A_CAR	CONTROL_MY_HOME	FIND_A_DENTIST
CAPTURE_A_PANORAMA	CREATE_A_3D_MODEL	FIND_A_DOCTOR
CHECK_ME_IN	EXPLORE_NEARBY	FIND_A_FLIGHT

...and so on.

1. https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/VoiceTriggers.Command

These don't require that you have the com.google.android.glass.permission.DEVELOPMENT permission that we've always set in the manifest up to now. Only these commands are officially sanctioned by Google, so if you have plans of someday deploying your application in the Google play store, you'll have to use them. The benefit of using these commands is that they are automatically internationalized into all languages supported by Glass.

An Activity with a Splash Screen

Our GameActivity is not too different from other activities we've written. There's an onCreate method that populates all of the fields that our app will use. We'll talk about UserInputManager, frameBuffer, GameEngine and RenderView in due course. For now, let's focus on the loadingCardView() method and the Handler object.

chapter-14/MissPitts/src/glass/misspitts/GameActivity.java

```
public class GameActivity extends Activity {
  public void onCreate( Bundle savedInstanceState ) {
    super.onCreate( savedInstanceState );
    setContentView( loadingCardView() );
    inputManager = new UserInputManager( this );
    final Bitmap frameBuffer = Bitmap.createBitmap( 640, 360, Config.RGB_565 );
    engine = new GameEngine( this, inputManager, frameBuffer );
    renderView = new RenderView( GameActivity.this, engine, frameBuffer );
    // After 3 seconds, switch the content view to the rendered game
    Handler handler = new Handler();
    handler.postDelayed(new Runnable() {
      @Override
      public void run() {
        setContentView( renderView );
      }
    }, 3000);
  }
```

First, let's look at the loadingCardView. We want to create a splash card that displays the name of the game (and the high-score, which we'll cover later) for a few seconds before starting the game. This allows the player a few seconds to get into the right frame of mind before starting the game, and it's also a good way to convey other information, like the high score, or maybe how to control the game.

Our splash screen will look something like this.

We're using a reference drawable called loading.png as a full background image, and overwriting the last high-score over top of it. We use the Card class as a convenient way to build a card view, but we could have just as easily made a layout with an ImageView and some text.

chapter-14/MissPitts/src/glass/misspitts/GameActivity.java

```
private View loadingCardView() {
  Card loadingCard = new Card( this );
  loadingCard.setImageLayout( ImageLayout.FULL );
  loadingCard.addImage( R.drawable.loading );
  loadingCard.setText(
      getResources().getString(R.string.high_score) + getSavedScores() );
  View cardView = loadingCard.getView();
  cardView.setKeepScreenOn( true );
  return cardView;
}
```

Before returning the view generated by the card, we set setKeepScreenOn(true) for good measure. This keeps the screen from dimming/pausing within the three seconds it takes for the game to start. We set this view at the start of onCreate.

If you look closely, the code calls setContentView twice—once immediately, and again inside of an anonymous Runnable class.

A Handler in Android manages the processing of messages in a queue. This queue can either act as a scheduler to run some code at a future date, or it can be a way to offload work to a different thread. We're using it in the first way, since we didn't pass in another thread's Looper.[2]

This is a low-tech way of creating a timed splash card. Our application is still loading in the background, and for three seconds the user will only see the

2. http://developer.android.com/reference/android/os/Looper.html

view constructed by the loadingCard method. Once the three seconds have passed, however, renderView becomes the new content view, overwriting the splash card.

With our basic application shell and splash screen underway, let's dig into some of those fields we skipped, starting with UserInputManager which, as you may have guessed from the name, manages user inputs.

Gesture and Sensor Inputs

A game drawing itself and playing music without user interaction is, frankly, just a cartoon. A game must provide an interface so a player can interact with the environment in some way. Moreover, this interface must be easy and natural. Otherwise, no matter how good your game's story or graphics may be, discomfort will stop anyone from playing it. So let's investigate our user control options, with an eye on playability.

For simplicity, we'll handle all player inputs through a class called UserInputManager. This class is in two parts—the first handles inputs that cause Miss Pitts to jump, the other handles making her walk, run, or stop.

One of the cooler features of Glass is its variety of inputs. We must forego the kind of hand-held controller you get in game consoles, or the variety of keyboards/joysticks on a PC. Let's consider the two character movements in our game: jumping and walking.

Tap Gesture

We'll start with making our character jump. You could interface with a variety of inputs, each with their own strengths and weaknesses. Voice is always an option, simply saying out loud the word "jump." It's hands free, but also, limits where you play the game. You can play in the privacy of your own home, or risk annoying people by talking aloud to yourself in a public bus. Voice also always has a lag, and reaction time between talking and an action happening is slower than other muscle movements, like twitching a finger.

Another option is to take advantage of the Glass sensors, where a player could flick his or her head to cause Miss Pitts to jump. That would also be hands-free and silent, but asking users to whip their head up repeatedly is a recipe for a hurt neck.

Instead, let's have the player tap the gesture bar to trigger a jump. We can actually track how many fingers are used to tap, so let's take advantage of that fact where one finger represents a short jump, two finger taps for higher jumps, and three for the highest.

We'll send gesture events onto our UserInputManager that implements GestureDetector.BaseListener, which requires this class to override onGesture(Gesture). We'll go over the SensorEventListener methods later.

chapter-14/MissPitts/src/glass/misspitts/UserInputManager.java

```
public class UserInputManager
  implements GestureDetector.BaseListener, SensorEventListener
{
  enum GameEvent {
    JUMP, JUMP_HIGHER, JUMP_HIGHEST,
    WALK, RUN, STOP
  }
  public UserInputManager( Context context ) {
    this.context = context;
    events = new ArrayList<GameEvent>(4);
    eventsBuffer = new ArrayList<GameEvent>(4);
    gestureDetector = new GestureDetector( context ).setBaseListener( this );
  }
  public synchronized List<GameEvent> getEvents() {
    events.clear();
    events.addAll( eventsBuffer );
    eventsBuffer.clear();
    return events;
  }
  public boolean dispatchGenericFocusedEvent( MotionEvent event ) {
    return gestureDetector != null && gestureDetector.onMotionEvent(event);
  }
  @Override
  public boolean onGesture( Gesture gesture ) {
    switch( gesture ) {
    case TAP:
      eventsBuffer.add( GameEvent.JUMP );         return true;
    case TWO_TAP:
      eventsBuffer.add( GameEvent.JUMP_HIGHER );  return true;
    case THREE_TAP:
      eventsBuffer.add( GameEvent.JUMP_HIGHEST ); return true;
    default:
      return false;
    }
  }
  // ...
```

Some of this might look familiar, like creating a new GestureDetector and setting this as it's BaseListener. There's also the dispatchGenericFocusedEvent method. Both of these are similar code to Chapter 12, *Advanced Rendering and Navigation*, on page ?, where we captured gestures via BalloonCountActivity, and shared those events with BalloonCountScrollView via dispatchGenericFocusedEvent.

The difference here is that we capture gestures, and rather than react immediately to them, we store corresponding game events in a buffer. So, for example, when a player makes a TWO_TAP Gesture (meaning, with two fingers), we save that as a JUMP_HIGHER game event.

Later, we'll see how the GameEngine periodically translates these events into game actions by calling getEvents. In this method, we synchronously copy the buffer into an unmodifiable list and flush the buffer. This allows us to continue accepting inputs from the player into the event buffer while the GameEngine iterates on new events.

Notice that our enum GameEvent has placeholders for all three jump heights. There are also values for walk, run, and stop. But how do we get those events?