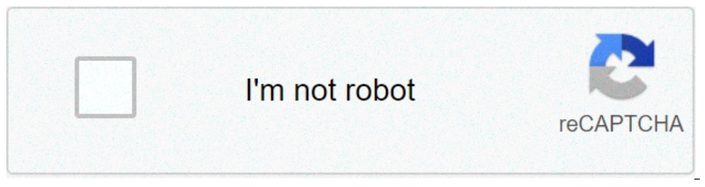

Continue

Bounce tales 7 java game for android

There are plenty of ways to create a game for Android and one important way is to do it from scratch in Android Studio with Java. This gives you the maximum control over how you want your game to look and behave and the process will teach you skills you can use in a range of other scenarios too – whether you're creating a splash screen for an app or you just want to add some animations. With that in mind, this tutorial is going to show you how to create a simple 2D game using Android Studio and the Java. You can find all the code and resources at Github if you want to follow along. In order to create our game, we're going to need to deal with a few specific concepts: game loops, threads and canvases. To begin with, start up Android Studio. If you don't have it installed then check out our full introduction to Android Studio, which goes over the installation process. Now start a new project and make sure you choose the 'Empty Activity' template. This is a game, so of course you don't need elements like the FAB button complicating matters. The first thing you want to do is to change AppCompatActivity to Activity. This means we won't be using the action bar features.Similarly, we also want to make our game full screen. Add the following code to onCreate() before the call to setContentView(): CodegetWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_FULLSCREEN); this.requestWindowFeature(Window.FEATURE_NO_TITLE); Note that if you write out some code and it gets underlined in red, that probably means you need to import a class. In other words, you need to tell Android Studio that you wish to use certain statements and make them available. If you just click anywhere on the underlined word and then hit Alt+Enter, then that will be done for you automatically! Creating your game view You may be used to apps that use an XML script to define the layout of views like buttons, images and labels. This is what the line setContentView is doing for us. But again, this is a game meaning it doesn't need to have browser windows or scrolling recycler views. Instead of that, we want to show a canvas instead. In Android Studio a canvas is just the same as it is in art: it's a medium that we can draw on. So change that line to read as so: CodesetContentView(new GameView(this)) You'll find that this is once again underlined red. But now if you press Alt+Enter, you don't have the option to import the class. Instead, you have the option to create a class. In other words, we're about to make our own class that will define what's going to go on the canvas. This is what will allow us to draw to the screen, rather than just showing ready-made views. So right click on the package name in your hierarchy over on the left and choose New > Class. You'll now be presented with a window to create your class and you're going to call it GameView. Under SuperClass, write: android.view.SurfaceView which means that the class will inherit methods – its capabilities – from SurfaceView.In the Interface(s) box, you'll write android.view.SurfaceHolder.Callback. As with any class, we now need to create our constructor. Use this code: Codeprivate MainThread thread; public GameView(Context context) { super(context); getHolder().addCallback(this); } Each time our class is called to make a new object (in this case our surface), it will run the constructor and it will create a new surface. The line 'super' calls the superclass and in our case, that is the SurfaceView. By adding Callback, we're able to intercept events. Now override some methods: Code@Override public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) { } @Override public void surfaceCreated(SurfaceHolder holder) { } @Override public void surfaceDestroyed(SurfaceHolder holder) { } These basically allow us to override (hence the name) methods in the superclass (SurfaceView). You should now have no more red underlines in your code. Nice.You just created a new class and each time we refer to that, it will build the canvas for your game to get painted onto. Classes create objects and we need one more. Creating threads Our new class is going to be called MainThread. And its job will be to create a thread. A thread is essentially like a parallel fork of code that can run simultaneously alongside the main part of your code. You can have lots of threads running all at once, thereby allowing things to occur simultaneously rather than adhering to a strict sequence. This is important for a game, because we need to make sure that it keeps on running smoothly, even when a lot is going on. Create your new class just as you did before and this time it is going to extend Thread. In the constructor we're just going to call super(). Remember, that's the super class, which is Thread, and which can do all the heavy lifting for us. This is like creating a program to wash the dishes that just calls washingMachine().When this class is called, it's going to create a separate thread that runs as an offshoot of the main thing. And it's from here that we want to create our GameView. That means we also need to reference the GameView class and we're also using SurfaceHolder which is contains the canvas. So if the canvas is the surface, SurfaceHolder is the easel. And GameView is what puts it all together. The full thing should look like so: Codepublic class MainThread extends Thread { private SurfaceHolder surfaceHolder; private GameView gameView; public MainThread(SurfaceHolder surfaceHolder, GameView gameView) { super(); this.surfaceHolder = surfaceHolder; this.gameView = gameView; } } Schweet. We now have a GameView and a thread! Creating the game loop We now have the raw materials we need to make our game, but nothing is happening. This is where the game loop comes in. Basically, this is a loop of code that goes round and round and checks inputs and variables before drawing the screen. Our aim is to make this as consistent as possible, so that there are no stutters or hiccups in the framerate, which I'll explore a little later.For now, we're still in the MainThread class and we're going to override a method from the superclass. This one is run. And it goes a little something like this: Code@Override public void run() { while (running) { canvas = null; try { canvas = this.surfaceHolder.lockCanvas(); synchronized(surfaceHolder) { this.gameView.update(); this.gameView.draw(canvas); } } catch (Exception e) {} finally { if (canvas != null) { try { surfaceHolder.unlockCanvasAndPost(canvas); } catch (Exception e) { e.printStackTrace(); } } } } You'll see a lot of underlining, so we need to add some more variables and references. Head back to the top and add: Codeprivate SurfaceHolder surfaceHolder; private GameView gameView; private boolean running; public static Canvas canvas; Remember to import Canvas. Canvas is the thing we will actually be drawing on. As for 'lockCanvas', this is important because it is what essentially freezes the canvas to allow us to draw on it. That's important because otherwise, you could have multiple threads attempting to draw on it at once. Just know that in order to edit the canvas, you must first lock the canvas. Update is a method that we are going to create and this is where the fun stuff will happen later on. The try and catch meanwhile are simply requirements of Java that show we're willing to try and handle exceptions (errors) that might occur if the canvas isn't ready etc. Finally, we want to be able to start our thread when we need it. To do this, we'll need another method here that allows us to set things in motion. That's what the running variable is for (note that a Boolean is a type of variable that is only ever true or false). Add this method to the MainThread class: Codepublic void setRunning(boolean isRunning) { running = isRunning; } But at this point, one thing should still be highlighted and that's update. This is because we haven't created the update method yet. So pop back into GameView and now add method. Codepublic void update() { } We also need to start the thread! We're going to do this in our surfaceCreated method: Code@Override public void surfaceCreated(SurfaceHolder holder) { thread.setRunning(true); thread.start(); } We also need to stop the thread when the surface is destroyed. As you might have guessed, we handle this in the surfaceDestroyed method. But seeing as it can actually take multiple attempts to stop a thread, we're going to put this in a loop and use try and catch again. Like so: Code@Override public void surfaceDestroyed(SurfaceHolder holder) { boolean retry = true; while (retry) { try { thread.setRunning(false); thread.join(); } catch (InterruptedException e) { e.printStackTrace(); } retry = false; } } And finally, head up to the constructor and make sure to create the new instance of your thread, otherwise you'll get the dreaded null pointer exception! And then we're going to make GameView focusable, meaning it can handle events. Codethread = new MainThread(getHolder(), this); setFocusable(true); Now you can finally actually test this thing! That's right, click run and it should actually run without any errors. Prepare to be blown away!It's… it's… a blank screen! All that code. For a blank screen. But, this is a blank screen of opportunity. You've got your surface up and running with a game loop to handle events. Now all that's left is make stuff happen. It doesn't even matter if you didn't follow everything in the tutorial up to this point. Point is, you can simply recycle this code to start making glorious games! Doing a graphics Right, now we have a blank screen to draw on, all we need to do is draw on it. Fortunately, that's the simple part. All you need to do is to override the draw method in our GameView class and then add some pretty pictures: Code@Override public void draw(Canvas canvas) { super.draw(canvas); if (canvas != null) { canvas.drawColor(Color.WHITE); Paint paint = new Paint(); paint.setColor(Color.rgb(250, 0, 0)); canvas.drawRect(100, 100, 200, 200, paint); } } Run this and you should now have a pretty red square in the top left of an otherwise-white screen. This is certainly an improvement.You could theoretically create pretty much your entire game by sticking it inside this method (and overriding onTouchEvent to handle input) but that wouldn't be a terribly good way to go about things. Placing new Paint inside our loop will slow things down considerably and even if we put this elsewhere, adding too much code to the draw method would get ugly and difficult to follow. Instead, it makes a lot more sense to handle game objects with their own classes. We're going to start with one that shows a character and this class will be called CharacterSprite. Go ahead and make that. This class is going to draw a sprite onto the canvas and will look like so Codepublic class CharacterSprite { private Bitmap image; public CharacterSprite(Bitmap bmp) { image = bmp; } public void draw(Canvas canvas) { canvas.drawBitmap(image, 100, 100, null); } } Now to use this, you'll need to load the bitmap first and then call the class from GameView. Add a reference to private CharacterSprite characterSprite and then in the surfaceCreated method, add the line: CodecharacterSprite = new CharacterSprite(BitmapFactory.decodeResource(getResources(),R.drawable.avdgreen)); As you can see, the bitmap we're loading is stored in resources and is called avdgreen (it was from a previous game). Now all you need to do is pass that bitmap to the new class in the draw method with: CodecharacterSprite.draw(canvas); Now click run and you should see your graphic appear on your screen! This is BeeBoo. I used to draw him in my school textbooks.What if we wanted to make this little guy move? Simple: we just create x and y variables for his positions and then change these values in an update method. So add the references to your CharacterSprite and then then draw your bitmap at x, y. Create the update method here and for now we're just going to try: Each time the game loop runs, we'll move the character down the screen. Remember, y coordinates are measured from the top so 0 is the top of the screen. Of course we need to call the update method in CharacterSprite from the update method in GameView.Press play again and now you'll see that your image slowly traces down the screen. We're not winning any game awards just yet but it's a start! Okay, to make things slightly more interesting, I'm just going to drop some 'bouncy ball' code here. This will make our graphic bounce around the screen off the edges, like those old Windows screensavers. You know, the strangely hypnotic ones. Codepublic void update() { x += xVelocity; y += yVelocity; if ((x & gt; screenWidth - image.getWidth()) || (x & lt; 0)) { xVelocity = xVelocity * -1; } if ((y & gt; screenHeight - image.getHeight()) || (y & lt; 0)) { yVelocity = yVelocity * -1; } } You will also need to define these variables: Codeprivate int xVelocity = 10; private int yVelocity = 5; private int screenWidth = Resources.getSystem().getDisplayMetrics().widthPixels; private int screenHeight = Resources.getSystem().getDisplayMetrics().heightPixels; There is plenty more to delve into here, from handling player input, to scaling images, to managing having lots of characters all moving around the screen at once. Right now, the character is bouncing but if you look very closely there is slight stuttering. It's not terrible but the fact that you can see it with the naked eye is something of a warning sign. The speed also varies a lot on the emulator compared to a physical device. Now imagine what happens when you have tons going on on the screen at once! There are a few solutions to this problem. What I want to do to start with, is to create a private integer in MainThread and call that targetFPS. This will have the value of 60. I'm going to try and get my game to run at this speed and meanwhile, I'll be checking to ensure it is. For that, I also want a private double called averageFPS. I'm also going to update the run method in order to measure how long each game loop is taking and then to pause that game loop temporarily if it is ahead of the targetFPS. We're then going to calculate how long it now took and then print that so we can see it in the log. Code@Override public void run() { long startTime; long timeMillis; long waitTime; long totalTime = 0; int frameCount = 0; long targetTime = 1000 / targetFPS; while (running) { startTime = System.nanoTime(); canvas = null; try { canvas = this.surfaceHolder.lockCanvas(); synchronized(surfaceHolder) { this.gameView.update(); this.gameView.draw(canvas); } } catch (Exception e) { } finally { if (canvas != null) { try { surfaceHolder.unlockCanvasAndPost(canvas); } catch (Exception e) { e.printStackTrace(); } } } timeMillis = (System.nanoTime() - startTime) / 1000000; waitTime = targetTime - timeMillis; try { this.sleep(waitTime); } catch (Exception e) {} totalTime += System.nanoTime() - startTime; frameCount++; if (frameCount == targetFPS) { averageFPS = 1000 / ((totalTime / frameCount) / 1000000); frameCount = 0; totalTime = 0; System.out.println(averageFPS); } } } Now our game is attempting to lock it's FPS to 60 and you should find that it generally measures a fairly steady 58-62 FPS on a modern device. On the emulator though you might get a different result.Try changing that 60 to 30 and see what happens. The game slows down and it should now read 30 in your logcat. Closing Thoughts There are some other things we can do to optimize performance too. There's a great blog post on the subject here. Try to refrain from ever creating new instances of Paint or bitmaps inside the loop and do all initializing outside before the game begins.If you're planning on creating the next hit Android game then there are certainly easier and more efficient ways to go about it these days. But there are definitely still use-case scenarios for being able to draw onto a canvas and it's a highly useful skill to add to your repertoire. I hope this guide has helped somewhat and wish you the best of luck in your upcoming coding ventures! Next – A beginner's guide to Java Android DevelopmentAndroid Studio, App development, Java