Android isometric game tutorial

I'm not robot

reCAPTCHA

Continue

I'm not robot

reCAPTCHA

In this mini-project, we'll draw a moving, controlled sprite to your Android device. We will draw Bob on the screen and when the screen is touched he will move to the right and when the screen is released he will stop. However, we have achieved much more than this as this simple 2D game engine will be easy to upgrade to handle many other gaming features. You'll also be able to simply copy/paste most of the code to get our first real, playable game; that will be the next project. You will also be able to use this code as a springboard to get started with 2D games of your own design. Rates are higher up to 95% off - clicking on the ad above. Create a new Android Studio project and call it Simple Game Engine. Create an empty activity and call it SimpleGameEngine. Add the bob.png file (see below) to the drawn folder, as we did in the Drawing graphics demonstration. Here's a little pseudo-code so we can evaluate the structure of the real code we'll write soon. The real code is quite long and using just a quick glance can seem complicated, although it really isn't. This pseudo-code will clarify the situation a little and allow us to present and discuss the code in managed pieces. You can also skip the end of the article and see the code in full for a clear explanation and context. You can simply copy and paste the full completed code into your project and get the simple game engine running; then go back and read all about it. Also, note that a download bonuses for this and every future tutorial are available on an exclusive download area for my Patreon subscribers. Pseudo code is not real code ... SimpleGameEngine Class onCreativ method to initiate a new GameView object as a view / End onCreate GameView class / Internal class in SimpleGameEngine Setting class in a special method Designer Run method causes an update, draw and control frame rate Upgrade Method update method to resume the method: starts when the game starts and sets our method of pause flow: starts When the game is closed and stops our stream onTouchEvent method called Android when touching the screen player / / End GameView internal class // Back in SimpleGameEngine class 2 more methods on the methodResume called Android when the game starts and calls GameView resume onPause method called Android when the game shuts down and causes GameView pause / / / End of simpleGameEngine and all our code 12345678910111111314151516171819222222222252527 SimpleGame Class Oncreative method to initiate a new GameView facility install a new GameView object as a view / End onCreate GameView class / Internal class in SimpleGameEngine Setting class in a special method Run method triggers update, draw and managing frame rate update Method Summary Method: Starts when the game starts and sets our method of pause flow: starts, when the game is closed and stops our stream on the MethodTouchEvent, called Android, when the screen is touched by the player / End of the internal class GameView / Back in SimpleGameEngine class 2 more methods onResume method called Android when the game starts and calls GameView resume on the pause method called Android when the game turns off and causes GameView pauses / The end of SimpleGameEngine and all our code How it works. Keep in mind that although the methods run their code, starting with the first line; how methods are executed is determined by when they are called. So that's what happens to our ready-made game engine. The player clicks the icon on his device to play our game. Android does some things in the background that we don't need to worry about; then he calls our method onCreate. In onCreate we announce and initiate a new class that we will encode, called GameView. The last thing onCreate does is install our new GameView object as a presentation for the app. Once onCreate is complete, the Android operating system calls onResume (in the SimpleGameEngine class). This method calls the resume method we'll write in our new GameView class. Our resume method starts with a draw, which triggers the execution method. The start-up method triggers an update and a draw, which is essentially the functionality of our game. The operating system will constantly call run many times per second. Thus, the GameView class processes our entire game, particularly from the start method, until the user presses a button on their device (perhaps back or home), which results in the operating system performing the onPause method in the SimpleGameEngine class, which in turn calls the pause method in the GameView class and stops our flow. The launch method will no longer be called and our game will cease to function. In addition to the operating system calling our startup method, it will also call our onTouchEvent method (also in GameView) in the main/default stream. This is where we can code the answer to the player's touch. Coding is a simple class of game engine Here's the code for our SimpleGameEngine class with additional methods (onPause and onResume), as well as our GameView class left for now. Copy and paste this into your project. import android.app.Activity; import android.content.Context; import android.graphics.Bitmap; import android.graphics.BitmapFactory; Import android.graphics.Canvas; Import android.graphics.Color; Import android.graphics.Paint; Import android.os.Bundle; Import android.view.MotionEvent; Import android.view.SurfaceHolder; Import android.view.SurfaceView; SimpleGameEngine community class expands GameView will be a kind of game / It will also keep the logic of the game / and respond to the touch of the screen as well as GameView gameView; @Override protected void onCreate (Bundle savedInstanceState) : Initiate gameView and install it as a gameView setContentView (gameView); GameView class will go here/ Simpler SimpleGameEngine methods will go here on 12345678910111111111141516171718192022242525252272782830333333333333333 import android.app.app. activity;import android.content.Context;import android.graphics.Bitmap;imported android.graphics.BitmapFactory;import android.graphics.Canvas;import android.graphics.Color;import android.graphics.Paint;import android.os.Bundle;import android.util.Log; import android.view.MotionEvent;import android.view.view.view.view. SurfaceHolder;import android.view.SurfaceView;Public class SimpleGameEngine expands activity / / gameView will be a kind of game / It will also keep the logic of the game // and respond to the touch of the screen as well as GameView gameView; @Override protected void onCreate (Bundle savedInstanceState) - super.onCreate (savedInstanceState); Initiate gameView and install it as a gameView view - the new GameView (it); setContentView (gameView); The GameView class will be here / Simpler SimpleGameEngine methods will go here Above we'll add all the necessary imports we'll need and a GameView object called gameView. We then encode the onCreate method, which simply initiates our GameView object and sets it on the look of the entire game, which is what the player will see on the screen. Rates are higher up to 95% off - clicking on the ad above. Coding the GameView class Now we will encode all this important GameView class and we will do it in pieces so we can talk about what is going on as we progress. We'll add simple additional methods (onResume and onPause) to the SimpleGameEngine class when we close GameView. If you think of the Drawing Graphics demo, we've turned everything to an ImageView-type object. And we install the object using setContentView as a view for the app. We can't just install any old object to be a presentation for the app. Fortunately, Android has some ready-made classes that we can expand that provide us with everything we need. In the next block of code, you see that we're announcing our GameView class and using the extends keyword to give it all the functionality of Android SurfaceView. This not only means that we can easily draw graphics on it, as we did with ImageView, but we can also use it in setContentView. Also in the first line of code (except for comments) we use the implementation keyword, followed by Runnable, which means that our class implements Runnable, which gives us all functionality of the flow and use/redefine a startup method that will become a kind of de facto cycle for our entire game. We also announce some of the objects and variables that we will need in the game. Enter the code below where the commentary is ... GameView class will go here for points. We'll discuss variables and objects later. Here's our gameView implementation/ This is an internal class. Notice how the final closing of curly braces / is inside SimpleGameEngine / Notice we implement runnable so we have a / Flow and can override the startup method. The GameView class expands SurfaceView implements Runnable / This is our thread game Read and zero; It's new. We need SurfaceHolder / When we use Paint and Canvas in the thread / We'll soon see it in action in the drawing method. SurfaceHolder of our Owner; Boolean, which we will install and unset ... when the game works- or not. Volatile boolean play; Canvas and canvas canvas object paint; Paint paint; This variable tracks the speed of the game's long fps frame; This is used to help calculate FPS private long time It's aram; Declare a Bitmap Bitmap bitmapBob type object; Bob starts not moving boolean isMoving and false; It can walk at a speed of 150 pixels per second float walkSpeedPerSecond 150; It launches 10 pixels from the left float bobXPosition No. 10; 12345678910111111114151517171819202222232425 @Override public invalid mileage () - while (game) / Capture of current time in milliseconds in startFrameTime long System.currentTimeMillis(); Update frame update Draw a frame to draw (); Calculate fps this frame // We can use the result for // animation time and more. timeThisFrame - System.currentTimeMillis() - startFrameTime; If (timeThisFrame zgt; 0) - fps 1000 / timeThisFrame; Here's the update below. All we do is check if isMoving is true, and if it's us to increase the position of Bob X (bobXPosition) to walkSpeedPerSecond divided into FPS. This will make all kinds of initialization ready when we use our class for our primary purpose. Also in the signature of the method we see that this object is transmitted. We need this to initiate our bit card and call the SurfaceView class designer. In our design method, we see what we call SurfaceView with a super keyword and then initiate our objectsholder, paint, and bitmapBob. Enter the code below that we just and we'll look at our startup method. When we initialize (a new call) on gameView / This special designer method launches a public GameView (context context) / The next line of code asks the SurfaceView class to customize our our How kind. Super (context); Initiate our Owner and paint our Owner's objects and getHolder(); Paint and new paint Download Bob from his .png bitmapBob and BitmapFactory.decodeResource (this.getResources) file; Now we are adjacent to the center of the universe as far as our simple game engine is concerned. The method works. Remember that the operating system constantly triggers this method from when we start our flow right around when we stop it. So we can think of the startup method as an endless loop that will run until the game is closed. This is sometimes referred to as a game cycle or game main. In it, we wrap our code in a loop (game) because the flow can actually start before we are fully established. The first thing we do if the game is true is to get the system time in a thousandth of one to eight seconds and store it in the startFrameTime variable. Then we call the update and draw. These two methods will keep the logic and graphic code of our game. After that we get the current system time again, subtract startFrameTime from it and store the result in timeThisFrame. This is the number of milliseconds it takes to perform our update and methods to draw this frame. If the following statement ensures that the answer is not zero, and if it is not, we assign a result of 1000, divided by timeThisFrame to fps. These are our current footage per second. We'll see how we use that very soon. Enter the startup method below, and we can move on to discussing the upgrade method. @Override public invalid run () - while (game) / Capture of current time in milliseconds long startFrameTime - System.currentTimeMillis (); Update frame update Draw a frame to draw (); Calculate fps this frame // We can use the result for // Animation time and more. If (timeThisFrame zgt; 0) - fps - 1000 / timeThisFrame; No 12345678910111111114151515617181920212222224 @Override public invalid mileage () - while (game) / Capture of current time in milliseconds in startFrameTime long System.currentTimeMillis(); Update frame update Draw a frame to draw (); Calculate fps this frame // We can use the result for // animation time and more. timeThisFrame - System.currentTimeMillis () - startFrameTime; If (timeThisFrame zgt; 0) - fps 1000 / timeThisFrame; Here's the update below. All we do is check if isMoving is true, and if it's us to increase the position of Bob X (bobXPosition) to walkSpeedPerSecond divided into FPS. This will make the same speed of motion for Bob regardless of current frames per second (fps) the device is working on. That's all for the update. Everything that needs to be updated is coming here / In later projects we will have dozens (of massive) objects. We will also do other things like collision detection. public invalid update () / If Bob moves (player touches the screen) / Then move it to the right based on his target speed and current fps. If (isMoving) bobXPosition and bobXPosition walkSpeedPerSecond / fps); We seen all the drawing code before in the graphic drawing demonstration, but there are some extra lines of code and one subtle more difference as well. First, we try to get a lock on the surface of the picture, and if successful, we draw the surface of our canvas. If you look closely at the call canvas.drawBitmap, instead of providing a number for x coordinates we use bobXPosition. We've already seen in the update method how we manipulate this variable. The only thing we haven't seen, but we'll be very soon as we switch between isMoving be true or false. Finally, we draw everything to the screen, and unlock the surface of the picture ourHolder.unlockCanvasAndPost (canvas); 123456789101111113111314161718192021222222222526267282930 / Draw a recently updated public void draw screen of try and catch blocks are a requirement forced upon us because of the class flow design. We don't need to fully understand them to make games. The unit is a try code to perform and a catch unit that to do if the attempt unit fails. Not unlike, if not another. An important code for the pause method is when we call gameThread.join and set the game to be false, it will turn off our game flow. Resume sets the playback variable on the true one, initiates our flow, and then launches it. The game is currently working. If SimpleGameEngine activity is suspended/stopped // Turn off our flow. public void pause - the game - false; Try - gameThread.join(); - catch (InterruptedException e) - Log.e (Error:, thread joining); If SimpleGameEngine Activity is running then / Run our thread. public invalid resume () - the game - the truth; gameThread.start (); - 123456791011111114141516171718919 // If SimpleGameEngine Activity is suspended/stopped // off our thread. public void pause - the game - false; Try - gameThread.join(); - Catch (InterruptedException e) - Log.e (Error:, flow joining); If SimpleGameEngine Activity is running / run our stream. public invalid resume - game - true; gameThread is a new thread (it); gameThread.start(); Getting a player entering one additional advantage of the SurfaceView class, which we expanded to make GameView is that it allows us to override the onTouchEvent method. The onTouchEvent method is called directly by the operating system every time the screen is touched. As part of this method, we can use data transferred to the MotionEvent object to determine if the player is telling our game to do something. This method will be complicated depending on the management system you are trying to implement. Future game projects will be part of this method in much more detail. At this point, all we have to do is switch, as shown in the code below, and handle the two cases. ACTION_DOWN when the player touches the screen and ACTION_UP when they remove the finger from the screen. All we do is install isMoving either true or false for each of these cases, respectively, and we implemented our game management. The SurfaceView class is implemented in onTouchListener / So we can override this method and detect the touch of the screen. @Override boolean onTouchEvent (MotionEvent motionEvent) - Switch (motionEvent.getAction () - MotionEvent.ACTION_MASK) / The player touched the screen case MotionEvent.ACTION_DOWN: / Install isMoving so Bob moves to the method of updating isMoving and the truth; A break; The player removed his finger from the screen case MotionEvent.ACTION_UP: / Set isMoving so Bob does not move isMoving - false; A break; The return is true; / Now we are at the end of our internal GameView class, so add closing curly braces. This is the end of our internal GameView class / This is the end of our internal GameView class, finishing the main class Now we are back in the SimpleGameEngine class, where we are adding override methods on Resume and onPause that will be caused by the operating system when the game is running or stopped. All they do is call almost the namesake techniques in the GameView class to manage our theme and default our entire game. Enter the last block of code for our simple game engine. Be sure to do so as part of the final closing of the SimpleGameEngine bracket. This method is performed when the player starts the game @Override protected void onResume () @Override - super.onResume() Tell gameView to resume the method to perform gameView.resume (); Tell GameView pause method to perform gameView.pause 12345678910111121314151617 / This method is performed when a player launches the game @Override protected void on Resum () - super.onResume(); Tell gameView to resume gameView to run gameView.run gameView.resume; This method is performed when a player leaves the game @Override protected void onPause () - super.onPause (); Tell gameView pause method to perform gameView.pause Running a simple game engine now you can run the game now - false; Try - gameThread.join(); If SimpleGameEngine activity is backward :-D. Although a somewhat longer project than some of the previous ones; we have achieved even more than it may seem. Now we have a frame/engine that we can reuse over and over again. If each line of code isn't crystal clear, don't worry, because as the code expands and improves, we'll get to know what's going on. Like every future getting more advanced, we're also ingringinging more Java and Android concepts. Let's move on and make a full, playable game or you can add a scrolling parallax background for this simple engine. Full list of android.app.Activity import code; import android.content.Context; Import android.graphics.Bitmap; import android.graphics.BitmapFactory; Import android.graphics.Canvas; Import android.graphics.Color; Import android.graphics.Paint; Import android.os.Bundle; Import android.util.Log; import android.view.MotionEvent; import android.view.SurfaceHolder; Import android.view.SurfaceView; SimpleGameEngine's public class expands activity / GameView will be a kind of game / It will also keep the logic of the game / and respond to the touch of the screen as well as GameView gameView; @Override protected void onCreate (Bundle savedInstanceState) - super.onCreate (savedInstanceState); Initiate gameView and install it as a gameView view - the new GameView (it); setContentView (gameView); The GameView class will go here / Here's our implementation of GameView / It's an internal class. Notice how the final closing of curly braces / is inside SimpleGameEngine / Notice we implement runnable so we have a / thread and can override the startup method. The GameView class expands SurfaceView implements Runnable / This is our thread game Count and zero; It's new. We need SurfaceHolder / When we use Paint and Canvas in the thread / We'll soon see it in action in the drawing method. SurfaceHolder of our Owner; Boolean, which we will install and unset / when the game works- or not. Volatile boolean play; Canvas and canvas canvas object paint; Paint paint; This variable tracks the speed of the game's long fps frame; This is used to help calculate FPS private long time It's aram; Declare a Bitmap Bitmap bitmapBob type object; Bob starts not moving boolean isMoving and false; It walk at a speed of 150 pixels per second float walkSpeedPerSecond 150; It launches 10 pixels from the left float bobXPosition No. 10; When we initialize (call new()) on gameView/ This special constructor method launches a public GameView (context context) / The next line of code asks the SurfaceView class to customize our object. How kind. Super (context); Initiate our Owner and draw our Owner's objects and getHolder(); Paint - new Download Bob from his .png bitmapBob file - BitmapFactory.decodeResource (this.getResources) file; Set our boolean to true - game our Reproduction is true - @Override public invalid launch () - while (game) / Capture of current time in milliseconds in startFrameTime long startFrameTime - (); Update frame update Draw a frame to draw (); Calculate fps this frame // We can use the result for // / - startFrameTime; If (Timelt's 0) - fps 1000 / timeThisFrame; The question / Everything that needs to be updated is included here / In later projects we will have dozens (massives) of objects. We will also do other things like collision detection. public invalid update () / If Bob moves (player touches the screen) / Then move it to the right based on his target speed and current fps. Issue / Draw a recently updated scene of a public invalid draw () / Make sure that our drawing surface is valid or we will crash if (ourHolder.getSurface().) isValid ()) / - Block canvas ready to draw canvas, ourHolder.lockCanvas (); Draw the background color canvas.drawColor (Color.argb,255, 26. 128, 182)); Choose the color of the paint.setColor (Color.argb (255, 249, 129, 0)); Make the text a little more paint.setTextSize (45); Displaying current fps on canvas.drawText (FPS: fps, 20, 40, paint); Draw the bob on bobXPosition, 200 pixels canvas.drawBitmap (bitmapBob, bobXPosition, 200, paint); Draw everything to the screen / and unlock the surface of the picture ourHolder.unlockCanvasAndPost (canvas); If SimpleGameEngine activity is suspended/stopped // Turn off our flow. public void pause - the game - false; Try - gameThread.join(); - catch (InterruptedException e) - Log.e (Error:, thread joining). If SimpleGameEngine Activity is running, then / Start our stream. public invalid resume () - game - true; gameThread.start (); The SurfaceView class is implemented on onTouchListener / So we can override this method and detect the touch of the screen. @Override boolean onTouchEvent (MotionEvent motionEvent) - Switch (motionEvent.getAction) - MotionEvent.ACTION_MASK) / The player touched the screen case MotionEvent.ACTION_DOWN: / Install isMoving so Bob moves to the method of updating isMoving and the truth; A break; The player removed his finger from the screen case MotionEvent.ACTION_UP: / Set isMoving so Bob does not move isMoving - false; A break; This is the end of our internal GameView class / More simple GameEngine methods will go here/ This method is performed when the player starts the game @Override protected void onResume () - super.onResume (); Tell gameView to resume gameView to run gameView.resume; This method is performed when a player leaves the game @Override protected void onPause () - super.onPause (); Tell gameView pause method to perform gameView.pause import android.app.Activity;import android.content.Context;import android.graphics.Bitmap;import android.graphics.BitmapFactory;import android.graphics.Paint;import android.os.Bundle;import android.util.util.Log;import android.view.MotionEvent;import android.view.SurfaceHolder;import android.view.SurfaceView;import android.graphics.Canvas;import android.graphics.Color expands activity ... gameView will be watching the game/ It will also keep the logic of the game/ and respond to the touch of the screen, as well as GameView gameView; @Override protected void onCreate (Bundle savedInstanceState) - super.onCreate (savedInstanceState); Initiate gameView and install it as a gameView view - the new GameView (it); setContentView (gameView); The GameView class will go here / Here's our GameView implementation... It's an internal class. Notice how the final closing of curly braces / is inside SimpleGameEngine / Notice we implement runnable so we have a / thread and can override the startup method. The GameView class expands SurfaceView implements Runnable / This is our thread game Count and zero; It's new. We need SurfaceHolder / When we use Paint and Canvas in the thread / We'll soon see it in action in the drawing method. SurfaceHolder of our Owner; Boolean, which we will install and unset ... when the game works- or not. Volatile boolean play; Canvas and canvas canvas object paint; Paint paint; This variable tracks the speed of the game's long fps frame; This is used to help calculate FPS private long time It's aram; Declare a Bitmap Bitmap bitmapBob type object; Bob starts not moving boolean isMoving and false; It walk at a speed of 150 pixels per second float walkSpeedPerSecond 150; It launches 10 pixels from the left float bobXPosition No. 10; When we initialize (call new()) on gameView / This special constructor method launches a public GameView (context context) / The next line of code asks the SurfaceView class to customize our object. How kind. Super (context); Initiate our Owner and paint our Owner's objects and getHolder(); Paint - new Download Bob from his .png bitmapBob and BitmapFactory.decodeResource (this.getResources) file; Set our boolean to the true game() To play and the truth; - @Override public invalid launch () - while (game) / Capture of current time / timeThisFrame; Update frame update Draw a frame to draw (); Calculate fps this frame // We can use the result for // Animation time and / timeThisFrame zgt; 0) - fps 1000 / fps); - / Draw a recently updated scene of a public invalid draw () / Make sure that our drawing surface is valid or we will fall apart if (ourHolder.getSurface().) isValid () -/ Blocking canvas ready to paint canvas, ourHolder.lockCanvas (); Draw the background color canvas.drawColor (Color.argb (255, 26, 128, 182)); Choose the color of the paint.setColor (Color.argb (255, 249, 129, 0)) Make the text a little more paint.setTextSize (45); Displaying current fps on canvas.drawText (FPS: fps, 20, 40, paint); Draw the bob on bobXPosition, 200 pixels canvas.drawBitmap (bitmapBob, bobXPosition, 200, paint); Draw everything to ourHolder.unlockCanvasAndPost (canvas); If SimpleGameEngine activity is suspended/stopped // Turn off our flow. public void pause - the game - false; Try - gameThread.join(); - Catch (InterruptedException e) - Log.e (Error:, flow joining); If simpleGame Engine Activity is running, then / Start our stream. public invalid resume () - game - true; gameThread.start (); The SurfaceView class is implemented onTouchListener / So we can override this method and detect the touch of the screen. @Override boolean onTouchEvent (MotionEvent motionEvent) - Switch (motionEvent.getAction () - MotionEvent.ACTION_MASK) / The player touched the screen case MotionEvent.ACTION_DOWN: / Install isMoving so Bob moves to the method of updating isMoving and the truth; A break; The player removed the finger from the screen case MotionEvent.ACTION_UP: / Set isMoving so Bob does not move isMoving and false; A break; The return is true; The question is/ This is the end of our internal GameView class / Simpler methods of the Game will go here / This method performs when the player starts the game @Override protected void onResume () - super.onResume(); Tell gameView to resume gameView to run gameView.resume; This method is performed when a player leaves the game @Override protected void onPause () - super.onPause() Tell gameView pause method to perform gameView.pause PerformView.pause }}