Chapter 26: Performance Tips and Tricks

By Ray Wenderlich

[The formatitng for this section will be updated by Vicki]

Sprite Kit does an excellent job of optimizing your game under the hood for great performance. Behind the scenes, Sprite Kit automatically creates object pools, avoids drawing offscreen nodes, creates bitmap font atlases and much more, so your games are fast and efficient without you having to think too much about it.

However, as your games get more complicated, from time to time you will run into performance issues. The goal of this chapter is to show you a few problem areas you might encounter when developing your games, and give you some workaround tips and tricks.

In particular, you are going to work on a game called Bullet Storm. Here’s what it currently looks like:



Bullet Storm is a demo of a side-scrolling space shooter. It’s a good start to a space game, but the performance is terrible. As you can see from this screenshot, it is running at less than 1 frame per second (FPS) on an iPhone 5. This app would be better off as a slide show!

In this chapter, you will debug Bullet Storm to discover its performance issues and then you will solve them. By the time you’re done, Bullet Storm will be running nice and smooth, and you’ll be well equipped to give your own games a performance boost!

## Getting started

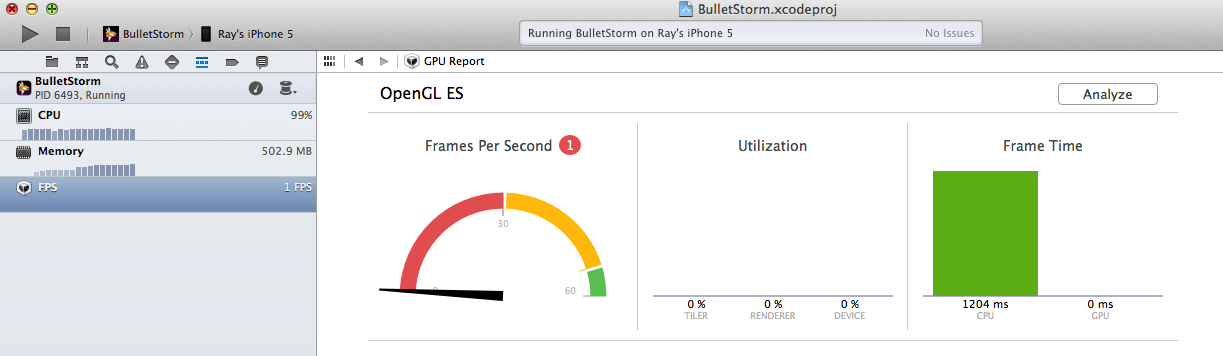
In the resources for this chapter, you will find a project called **BulletStorm-Starter**. Open the project and build and run on your iOS device.

**Note:** Remember that it’s important to run on your iOS device when testing for performance, as the artificial conditions of the Simulator will give inaccurate results.

You should see a very sluggish game appear (your colors and exact FPS may vary):



While the game is running, switch to the sixth navigator tab – the **Debug** **Navigator**. Click the FPS section on the left and you will see the game’s FPS (1FPS on my iPhone 5), as well as how many milliseconds your app is using the CPU or GPU each frame:



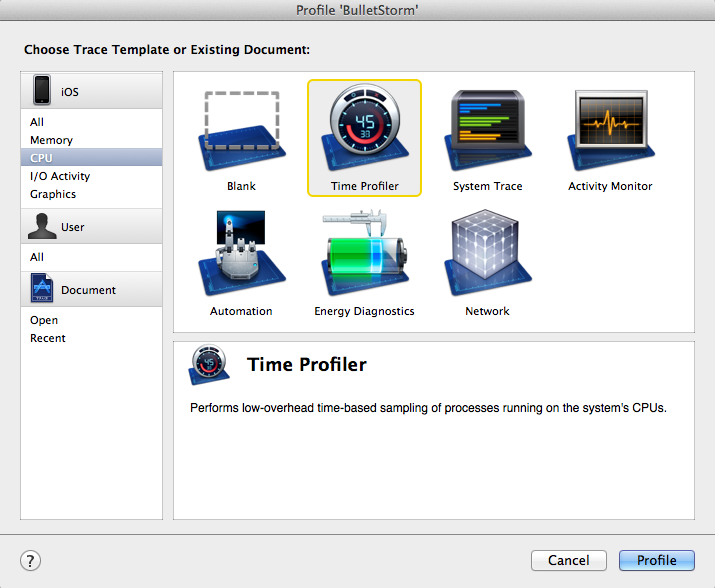
As you can see in this screenshot, an overwhelming majority of the time is spent on the CPU: 900-1,000 ms on my iPhone 5. This is generally a sign of a performance problem – you want your game to use the GPU as much as possible so that the graphics hardware can do the hard work.

Obviously something very wrong is happening here, but how can you find out what it is? Well, you could dig through the code looking for issues, but there’s a much better way to find performance problems, using your friend Instruments.

## Introducing Instruments

Instruments is a handy set of tools built into Xcode that you can use to peek into your app. Instruments has tools that can find memory leaks, check resource usage and – what you’re going to do in this section – analyze your app for performance.

Stop Bullet Storm and go to **Product\Profile** from the Main Menu (shortcut key ⌘I). The following pop-up will appear:

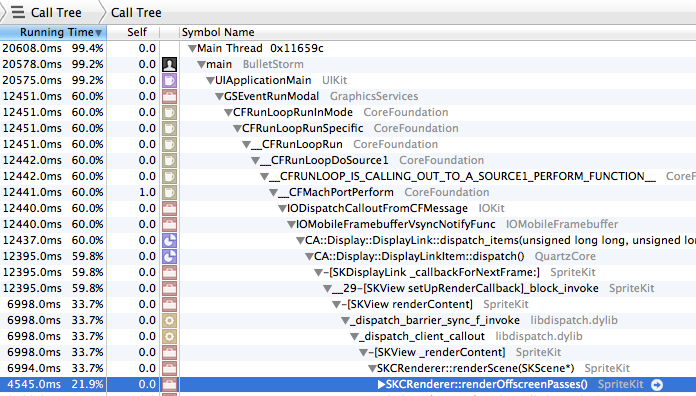


This displays the various instruments that you can integrate into your app. Here you want to find out what’s taking so long, so choose **CPU** under the **iOS** section on the left, and then choose **Time Profiler** and click **Profile**. A window will appear that looks like this:

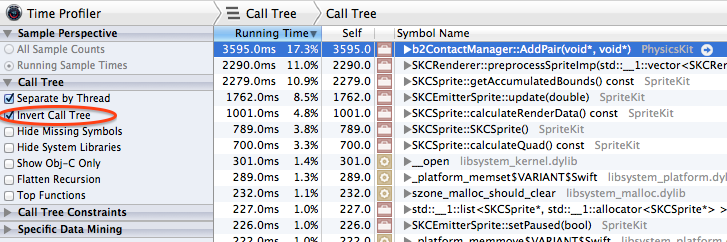


The top panel shows you a graph of the CPU use your app requires over time – notice the constant heavy usage!

The lower panel shows you a percentage-wise breakdown of where the CPU spends its time. You’ll see that the CPU is mostly occupied with a function called Main Thread. Click the down arrow next to Main Thread to display the list of sub-functions called by Main Thread and how much time the CPU spends on each. The panel sorts the sub-functions in order of running time, so keep opening the most expensive item until you dig deep enough to reveal functions that look related to Sprite Kit:



This is one way to gather clues about the problem, but there’s an even easier way. Usually you just want to see what “leaf” methods – that is, the ones at the end of the call tree – are taking up the largest amount of time. To figure this out, click the **Invert Call Tree** checkbox on the left:



This inverts the order of the tree so that the methods that take the most time are at the top of the tree, and their children are the callers of the methods.

Take a look at the first four methods in this list:

* b2ContactManager::AddPair. This method seems to be related to managing physics collisions. As a general rule, testing for collisions gets more expensive the more objects there are in the game. This gives you a hint to check Bullet Storm to see how many physics objects are in the game at one time – maybe there’s a way you can reduce them.
* SKRenderer::preprocessSpriteImpl and SKCSprite::getAccumulatedBounds. These methods seem to be related to processing sprites. As a general rule, the more sprites are in a game, the slower it performs. These methods suggest you should check Bullet Storm to see how many sprites are in the game at any one time – maybe there’s a way you can reduce them, too.
* SKCEmitterSprite::update. This method seems to be related to processing particle systems. As a general rule, particle systems get more expensive the more particles they generate. This tells you to check Bullet Storm to see how many particle systems are in the game and how many particles each generates, and see if you can find a way to reduce those.

If you are an advanced reader, this is a good chance to challenge yourself: Can you take things from here and find and resolve Bullet Storm’s performance problems? There have actually been plenty of hints about how to solve these issues in the earlier chapters of this book! ☺

But if you want to walk through it step-by-step, keep reading. Let’s start with the first two hints and peek into Bullet Storm to see if there is a way to reduce the number of physics bodies and sprites.

## Touring the code

1. If you haven’t already, spend a few minutes looking through Bullet Storm’s source code to see how it works. Here are a few notes:

* There is a class for each of the objects in the game: asteroids, lasers, explosions, enemies and the player. They all derive from a class called Entity.
* Most of the game logic is in MyScene. The code should look quite familiar – it is based on the example projects from other parts of this book!
* The game is already set up to use texture atlases (see **sprites.atlas**), and you learned about why that is so important in the previous chapter. If the game weren’t using texture atlases, its performance would be even worse. I know that’s hard to imagine! ☺

As you’re looking through the code, keep an eye out for any sprites or physics bodies that are created unnecessarily. If you think you’ve found the source of the problem, keep reading to the next section to see if you’re right.

**Hint**: Remember Zombie Conga!

## Reducing sprites and physics bodies

You may have come across the following function in **MyScene.m**:

- (void)spawnAsteroids

{

for (int i = 0; i < 500; ++i) {

Asteroid \*asteroid = [[Asteroid alloc]

initWithAsteroidType:arc4random\_uniform(NumAsteroidTypes)];

asteroid.name = @"asteroid";

asteroid.position =

CGPointMake(

self.size.width + asteroid.size.width/2 +

(i\*self.scene.size.width\*0.25),

RandomFloatRange(

asteroid.size.height/2,

self.size.height-asteroid.size.height/2));

[\_fgLayer addChild:asteroid];

}

}

This function creates an asteroid field through which the player must navigate. It creates a large number of asteroids (500 to be exact) offscreen to the right and moves them slowly to the left, across the screen.

This is similar to having a “level file” that specifies the position of each asteroid, and then creating each asteroid at the appropriate offscreen location on startup.

You might not see a problem here. This is an easy and straightforward way to create an asteroid field and Sprite Kit doesn’t draw nodes that aren’t visible on the screen.

However, even when nodes are offscreen, Sprite Kit still needs to do processing on the nodes: running actions, checking for offscreen collisions or processing any attached particle systems, for example. That’s the problem with the asteroids here – Sprite Kit has to perform many calculations on asteroids that aren’t even visible. Apple recommends that you only add a node to the scene graph when:

* It has a reasonably good chance of being rendered in the future.
* The node runs actions that are required for accurate gameplay.
* The node has a physics body that is required for accurate gameplay.

In the case of offscreen asteroids, none of these conditions apply. So you should be able to get much better performance by only creating asteroids when you need them – that is, right before they appear on the screen, just like in Zombie Conga!

To fix this, delete spawnAsteroids and replace it with the following:

- (void)spawnAsteroid

{

Asteroid \*asteroid = [[Asteroid alloc]

initWithAsteroidType:arc4random\_uniform(NumAsteroidTypes)];

asteroid.name = @"asteroid";

asteroid.position = CGPointMake(

self.size.width + asteroid.size.width/2,

RandomFloatRange(asteroid.size.height/2,

self.size.height-asteroid.size.height/2));

[\_fgLayer addChild:asteroid];

}

This is a helper method that creates an asteroid just offscreen to the right.

Then in initWithSize:, delete this line:

[self spawnAsteroids];

And add the following in its place:

[self runAction:[SKAction repeatActionForever:

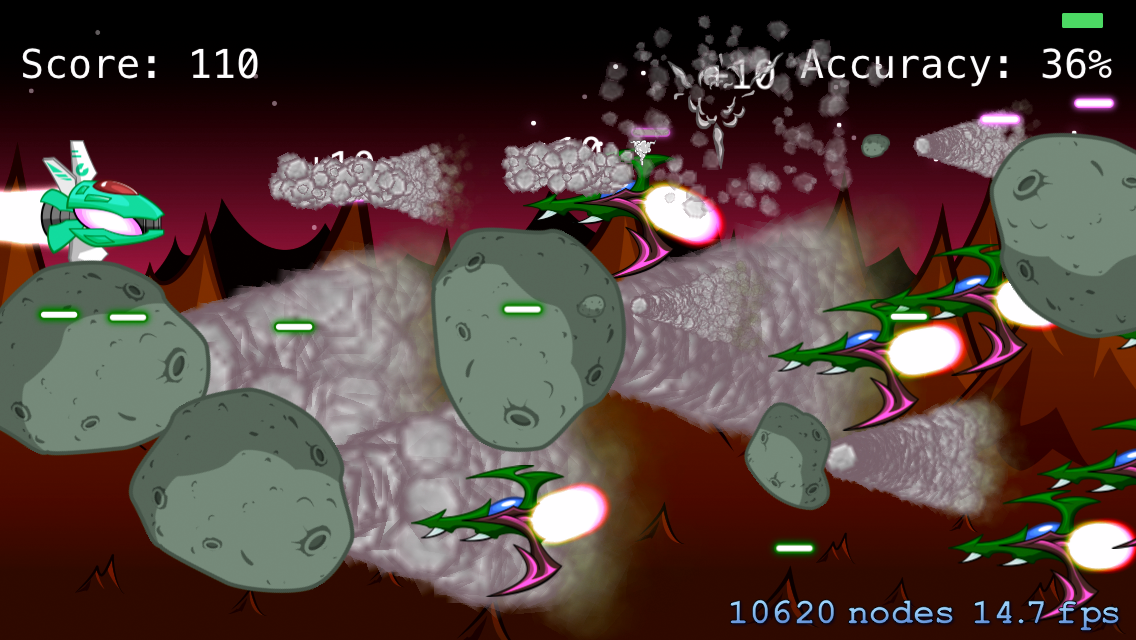
[SKAction sequence:@[

[SKAction performSelector:@selector(spawnAsteroid)

onTarget:self],

[SKAction waitForDuration:0.25]]]]];

This calls the helper function to spawn an asteroid every 0.25 seconds. Now build and run and you should see a marked performance increase:



From 1 FPS to 14.7 on my iPhone 5 – not a bad start!

However, 14.7 FPS still isn’t great, plus that node count of 10620 looks insane! The next step is to examine the particle systems to see if you can tune down anything there.

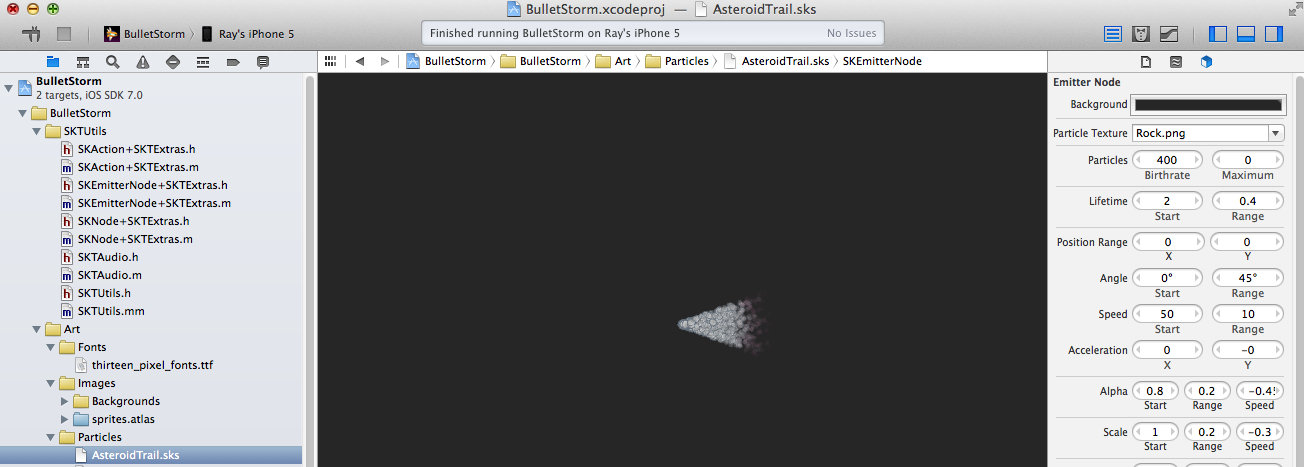
**Note**: Your frame rate and node count will probably vary wildly if you tilt your device to play the game rather than resting it on your desk. That’s mainly because your ship’s bullets will collide with more enemies if you move around the screen.

This is another great opportunity to challenge yourself: Can you tweak the particle systems so that they perform a bit better, but still look good? Try it out on your own, and if you get stuck, follow along with the next section!

## Reducing particle systems

Particle systems perform well in Sprite Kit and you should feel free to use them liberally in your games. However, since particle systems become more expensive as they generate greater numbers of particles, you want to make sure yours generate just enough to achieve the desired effect.

Let’s take a look at one of the particle systems in Bullet Storm. Open **BulletStorm\Art\Particles\AsteroidTrail.sks**, view the SKNode inspector in the **Utilities** section on the left, and you’ll see the following:



The important settings to notice are Particles Birthrate, Particles Maximum and Lifetime Start/Range. To review:

* **Particles Birthrate** represents how many particles are generated per second. In this instance, 400 particles are generated per second.
* **Particles Maximum** represents the maximum number of particles to generate at a time. Here it is set to 0, which means unlimited.
* **Particles Lifetime/Range** represents how long a particle lives before it is destroyed. Here it is set to 2 with a range of 0.4, meaning a particle will live from 1.6-2.4 seconds.

1. To determine the average number of particles onscreen for a given particle system, use this algorithm:

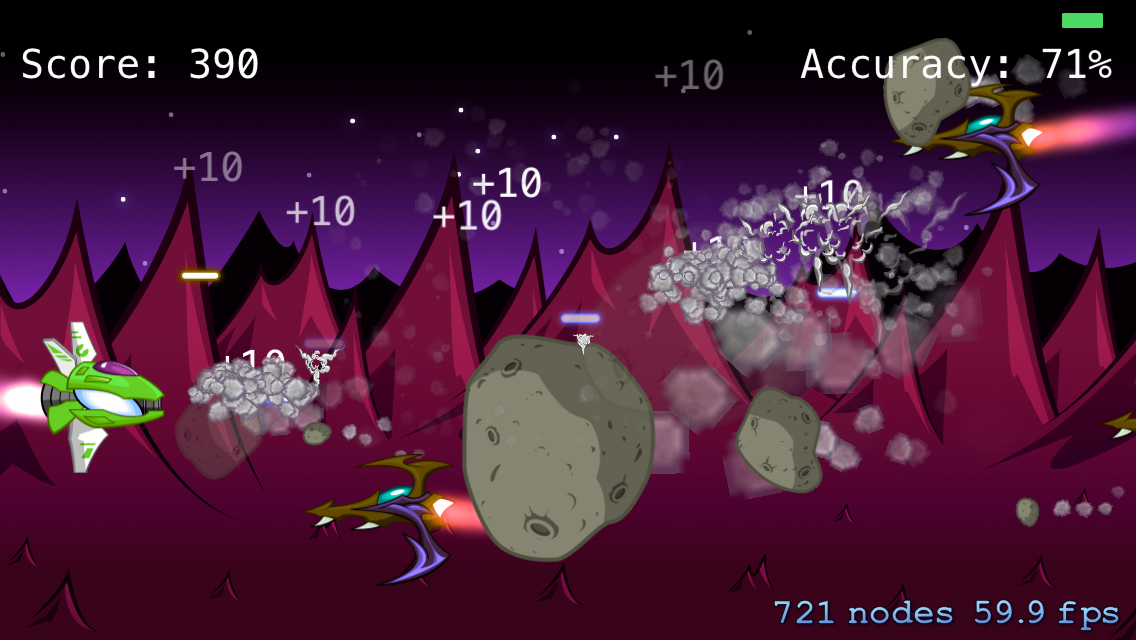
* Is **Particles Maximum** 0? Then the average equals **Particles Birthrate** x **Particles Lifetime**.
* Is **Particles Maximum** not 0? Then the average either equals **particles maximum** or **Particles Birthrate** x **Particles Lifetime**, whichever value is lower.

In this case, Particles Maximum is 0, so 400 x 2 = an average of 800 particles onscreen per second. That’s a lot for a simple asteroid trail effect – tone this way down by setting the **Particles Birthrate** to 4. Now it has on average 8 nodes per second, which will still result in a neat effect.

Repeat this for the other effects in the game:

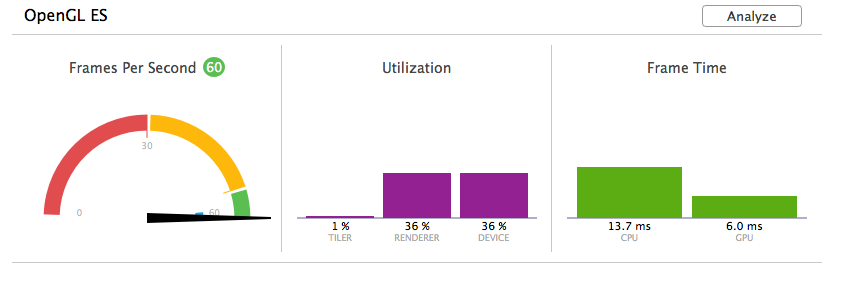
* For **EnemyTrail.sks**, set the **Particles Birthrate** from 2000 to **200**.
* For **Explosion.sks**, set the **Particles Birthrate** from 5000 to **500**.
* For **PlayerTrail.sks**, set the **Particles Birthrate** from 300 to **30**.

That’s a lot fewer particles! Build and run, and you’ll see another performance improvement:

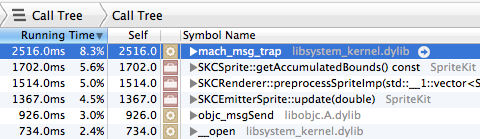


The game is almost up to 60 FPS on my iPhone 5 at this point. And it still looks great – some might say even better than before!

Switch to the **Debug Navigator** again (the sixth navigator tab), and look at the **FPS** section. You’ll see that now the game only spends 13.6 milliseconds using the CPU, and there’s a much better balance between CPU and GPU usage:



Now let’s check the profiler results. Stop the app, run it under the Time Profiler again and make sure **Invert Call Tree is checked**:



You will notice the following changes from the last time:

* b2ContactManager::AddPair is no longer on the list, which means that performance calculations are no longer part of the bottleneck. Just-in-time object spawning for the win!
* mach\_msg\_trap is on the list, which means the CPU is actually idle a fair percentage of the time. This is a good thing – it means you’re not using up all of the CPU!
* SKRenderer::preprocessSpriteImpl, SKCSprite::getAccumulatedBounds, and SKCEmitterSprite::update are still on the list, but a lot less so than before. That means your game is spending most of its time processing sprites and particle systems. If you needed to optimize still further, you could look for ways to use fewer sprites and particle systems – but for now, performance is pretty good.

Bullet Storm is running well enough that we’ll stop here. For your own games, you can run through a similar process using Instruments. However, your problems may differ from the ones just covered. The following section is an overview of other types of performance bottlenecks you may encounter, and gives you some tips and tricks for how to resolve them.

## Other performance tips and tricks

### Use effect nodes and Core Image sparingly

As mentioned in Chapter 12, “Effect Nodes and Core Image”, using effect nodes and Core Image in your games can result in some amazing effects, but they can also consume a lot of system resources.

Here are a few workarounds to consider if you have an effect node-related performance issue:

* **Are you filtering the lowest-possible granularity**? For example, applying a filter to the entire scene is a lot more expensive than applying a filter to a single small sprite or even a single layer of sprites. Make sure you’re only filtering exactly what you need to filter.
* **Can you rasterize the effect?** SKEffectNode has a property called shouldRasterize. This is set to NO by default, which means that each frame, Sprite Kit discards the previous result and re-applies the effect. If you set this to YES, it caches the image and reuses it until one of the SKEffectNode’s children nodes change. This is great to use if you’re applying an effect to something that doesn’t change often, like a background image – it can result in much faster performance.
* **Are there pre-rendered or “fake” alternatives?** In cases where you try the previous two tips and your game still doesn’t perform well enough, consider using an alternative to effect nodes. Is there any way you can pre-render the effect into a sprite that you ship with the app? Or fake it by some combination of pre-rendered sprites overlaid on top of each other?
* **Can you reduce the size of the effect?** In cases such as mask nodes, it often seems easiest to align your mask with your nodes by making them all the same size and then giving them all the same position. However, try to use as little transparent space as possible because transparent pixels add processing time. Instead, trim away any unused areas of your masks and adjust their positions to align them appropriately.

If you want to play around with effect node optimization, you’re in luck – a challenge awaits you at the end of this chapter!

### Object pooling

In the games in this book, whenever you need to create a new object (like a laser or an explosion effect), you simply create a new object at that time.

However, on iOS devices, it is expensive to allocate objects at runtime. One way to improve performance is to pre-allocate an array of objects and grab the next available object when you need it.

Note that Sprite Kit seems to do some object pooling for you behind the scenes, so it’s not as critical to implement object pooling yourself when using Sprite Kit as compared to other game frameworks. However, if you find that a large amount of your game’s time is spent in allocation routines, it might pay off to implement some object pooling.

If you want to play around with this, more good luck – there’s a challenge for that at the end of this chapter, too!

### Perform long-running operations in the background

By default, everything in Sprite Kit runs on the main thread. If you have a computationally intensive operation, such as a complicated pathfinding algorithm, you may wish to perform this on a background thread. There is an easy way to run a block of code on a background queue in Sprite Kit: through the runBlock:queue: action.

### Use texture atlases – and organize them wisely

This is probably the most important tip – which is why the book dedicates an entire chapter to the matter. If you haven’t already, be sure to read Chapter 25, “Performance: Texture Atlases.”

### Keep physics as simple as possible

As you saw with Bullet Storm, there is a cost associated with using the physics engine. Here are a few rules of thumb:

* **The more physics bodies, the more expensive for performance**. Consider making physics bodies only for sprites that actually need them. If a sprite doesn’t engage in collision or contact detection, then it doesn’t need a physics body. You may be able to design your gameplay in a way to reduce the requirements for physics bodies as well.
* **Prefer static physics bodies to dynamic ones.** That is, if you know an object will not move, set its dynamic property to NO. This allows the physics engine to perform various optimizations that increase performance.
* **The more vertices in your physics bodies, the more expensive for performance**. Consider making a simplified collision shape for your sprite – it doesn’t have to match the sprite’s shape exactly. You’ll find that the faster your objects move, the less exact the physics shape needs to be to give a visually satisfying collision. Remember, you’re trying to make a game, not a real-world physics simulation.
* **Use** usesPreciseCollisionDetection **sparingly**. This is a flag you can set on physics bodies to prevent fast-moving bodies from passing through other bodies within a single frame. It sounds good, but it does have a performance cost, so usually you want to set this only when you actually need it.

And that’s it for performance tips and tricks – and for the technical content of this book! Take a bow, take a break, and get ready for one last treat: learning how to make 2D art for your game.

But if you’ve got to have “one last bit of coding,” continue on with these challenges!

## Challenges

There are two challenges in this chapter – one to add some SKEffectNode improvements to Bullet Storm, and one to add some object pooling.

If you get stuck, you can find solutions in the resources for this chapter – but as always give them your best shot first!

### Challenge 1: SKEffectNode improvements

Currently Bullet Storm is applying a random hue adjust effect to the entire scene. Applying a filter to the entire scene is one of the most expensive things you can do with Core Image, and it’s not really necessary for this game. Your challenge is to modify the game so that it filters the background layer only.

Here are a few tips:

* In **MyScene.m** inside initWithSize:, comment out the lines that set the filter and shouldEnableEffects on the scene.
* Still inside initWithSize:, in the for loop that creates the background sprites, create a new SKEffectNode for each background sprite and call it bgParent.
* Set filter on bgParent to the Core Image filter.
* Set shouldEnableEffects on bgParent to YES.
* Set name on bgParent to @”bg”.
* Set the position of bgParent to be CGPointMake(i \* bg.size.width, 0).
* Set shouldRasterize on bgParent to YES.
* Add bg as a child of bgParent.
* Add bgParent as a child of bgLayer.
* Comment out the three older lines that set the bg’s position, and name and added it as a child of the bgLayer (leave the line that sets the zPosition).

Build and run, and you should have a version of the game that has faster SKEffectNode performance! For an even greater challenge, try to find proof of the improvement before and after using Instruments.

### Challenge 2: Laser object pooling

The objects that you create most frequently in Bullet Storm are lasers, so your challenge is to pre-allocate and reuse a bunch of laser sprites instead of continuously allocating them.

Here are a few hints on how to accomplish this:

* In **MyScene.m**, create a new function called spawnPlayerLasers. It should fill the premade \_playerLasers array with 100 laser objects. Then the function should set each laser object to hidden and add it to the \_fgLayer.
* Call spawnPlayerLasers in initWithSize:.
* At the beginning of spawnPlayerLaser, instead of creating a new laser each time, loop through the \_playerLasers array. If you find a laser that isn’t hidden, continue using that laser and set hidden to NO. Also comment out the line in that method that adds the laser to the \_fgLayer, since the laser should be there already.
* In **Laser.m’s** cleanup method, comment out the removeFromParent line. Instead, set physicsBody to nil and hidden to YES.

Build and run, and you should now have an even faster version of the game with object pooling! For an even greater challenge, try to find proof of the improvement before and after using Instruments.

And that’s a wrap! Congratulations for making it through to the end. Now it’s time to switch gears and get creative – it’s time to draw! ☺