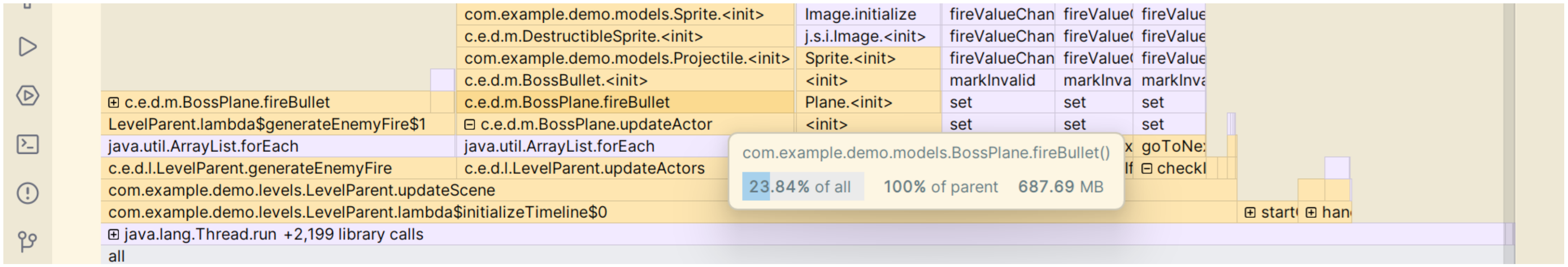
Evaluate Memory Benefit from Object Pooling by comparing:

- **“nopool” (branch: nopool)** – no object pooling (reuse BossBullet & EnemyBullet, handle off-screen)
- **“pooled” (branch: master)** – with object pooling (reuse BossBullet & EnemyBullet, handle off-screen)

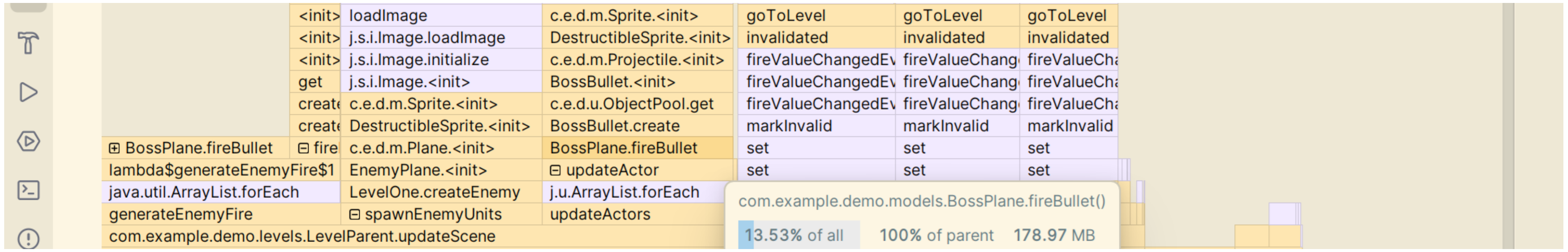
A) BossBullet from BossPlane.fireBullet()

- used 23.8% (688MB) of all memory for nopool
- vs only 13.5% (179MB) for pooled

nopool:


com.example.demo.models.Sprite.<init>
c.e.d.m.DestructibleSprite.<init>
com.example.demo.models.Projectile.<init>
c.e.d.m.BossBullet.<init>
⊞ c.e.d.m.BossPlane.fireBullet
c.e.d.m.BossPlane.fireBullet
LevelParent.lambda$generateEnemyFire$1
⊟ c.e.d.m.BossPlane.updateActor
java.util.ArrayList.forEach
java.util.ArrayList.forEach
c.e.d.l.LevelParent.generateEnemyFire
c.e.d.l.LevelParent.updateActors
com.example.demo.levels.LevelParent.updateScene
com.example.demo.levels.LevelParent.lambda$initializeTimeline$0
⊞ java.lang.Thread.run +2,199 library calls
all
Image.initialize
j.s.i.Image.<init>
Sprite.<init>
<init>
Plane.<init>
<init>
fireValueChan
fireValueChan
fireValueChan
markInvalid
set
set
fireValue(
fireValue(
fireValue(
markInva
set
set
fireValue
fireValue
fireValue
markInva
set
set
goToNe
⊟ checkI
⊞ start
⊞ han
com.example.demo.models.BossPlane.fireBullet()
23.84% of all
100% of parent
687.69 MB

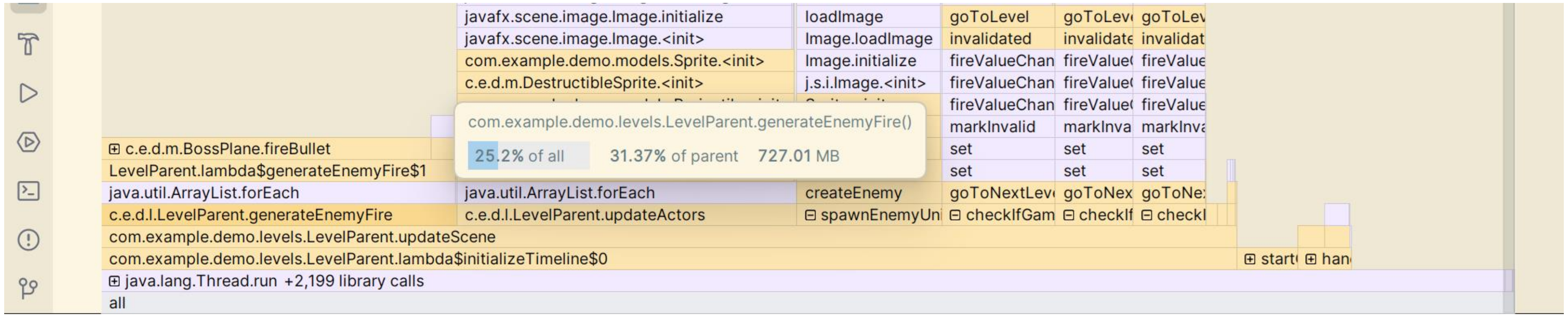
pooled:



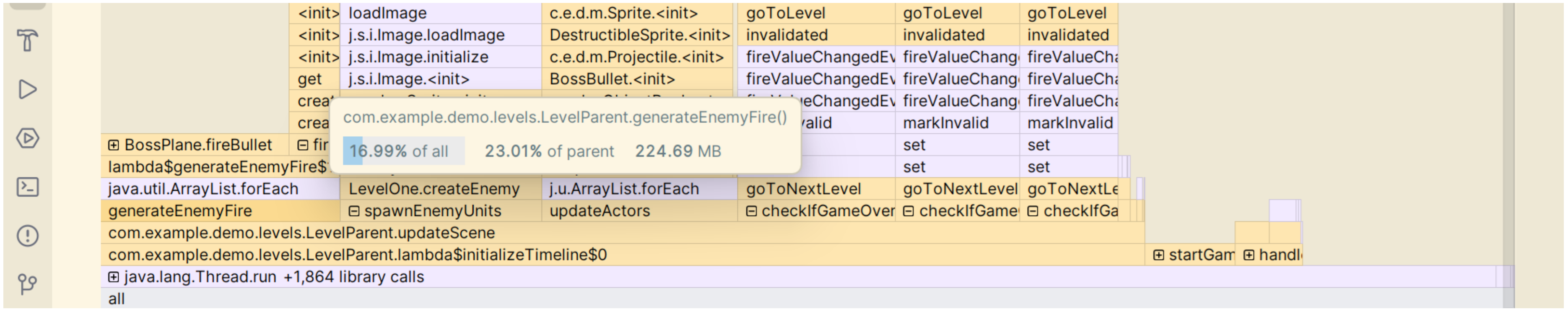
B) EnemyBullet from generateEnemyFire()

- used 25.2% (727MB) of all memory for nopool
- vs only 17.0% (225MB) for pooled

nopool:

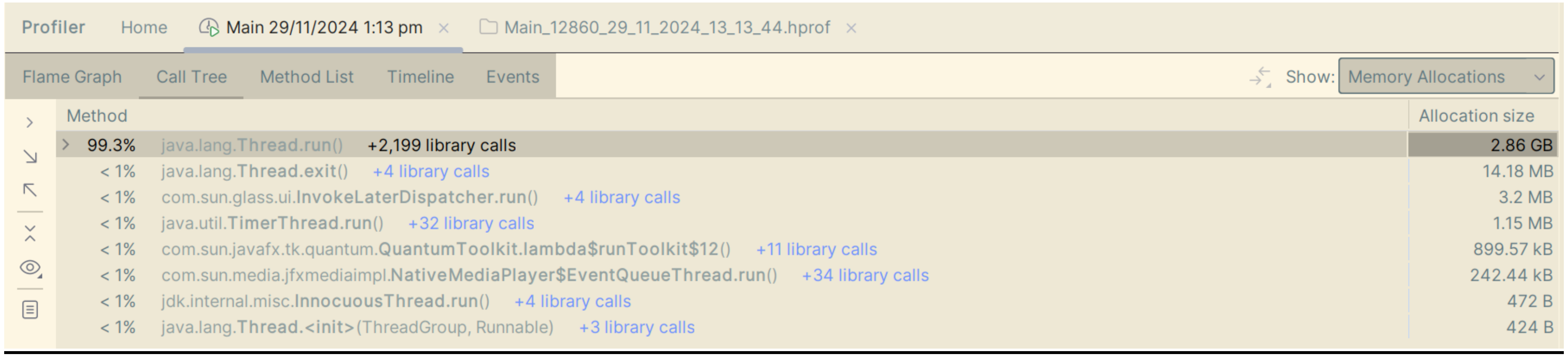

javafx.scene.image.Image.initialize
javafx.scene.image.Image.<init>
com.example.demo.models.Sprite.<init>
c.e.d.m.DestructibleSprite.<init>
loadImage
Image.loadImage
Image.initialize
j.s.i.Image.<init>
goToLevel
invalidated
fireValueChan
markInvalid
set
com.example.demo.levels.LevelParent.generateEnemyFire()
25.2% of all
31.37% of parent
727.01 MB
⊞ c.e.d.m.BossPlane.fireBullet
LevelParent.lambda$generateEnemyFire$1
java.util.ArrayList.forEach
c.e.d.l.LevelParent.generateEnemyFire
java.util.ArrayList.forEach
c.e.d.l.LevelParent.updateActors
createEnemy
⊟ spawnEnemyUni
goToNextLev
⊟ checkIfGam
com.example.demo.levels.LevelParent.updateScene
com.example.demo.levels.LevelParent.lambda$initializeTimeline$0
⊞ start
⊞ han
⊞ java.lang.Thread.run +2,199 library calls
all

pooled:



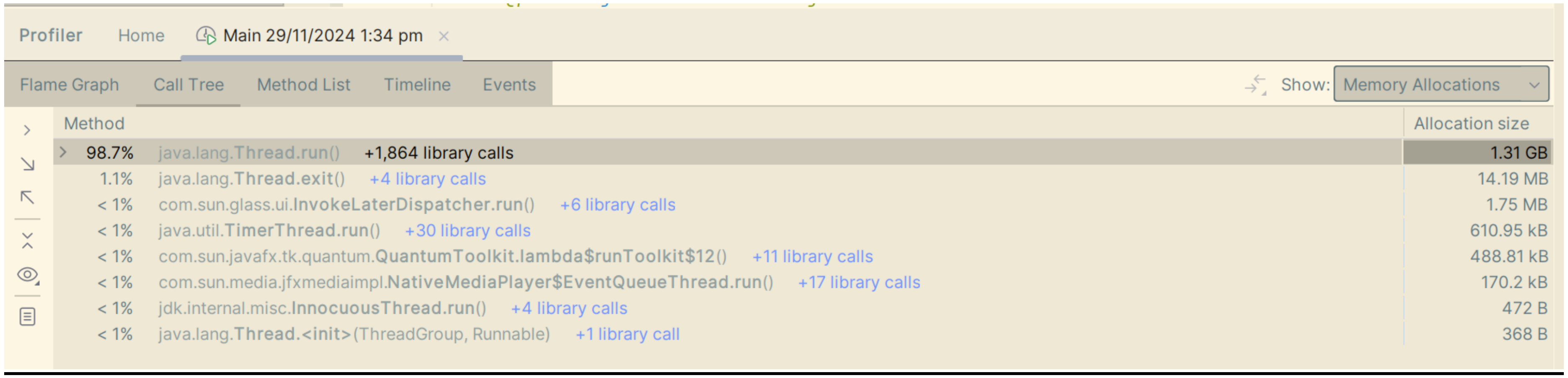
C) Call Tree Memory Allocation

nopool: 2.86GB



Method			Allocation size
99.3%	java.lang.Thread.run()	+2,199 library calls	2.86 GB
< 1%	java.lang.Thread.exit()	+4 library calls	14.18 MB
< 1%	com.sun.glass.ui.InvokeLaterDispatcher.run()	+4 library calls	3.2 MB
< 1%	java.util.TimerThread.run()	+32 library calls	1.15 MB
< 1%	com.sun.javafx.tk.quantum.QuantumToolkit.lambda$runToolkit$12()	+11 library calls	899.57 kB
< 1%	com.sun.media.jfxmediaimpl.NativeMediaPlayer$EventQueueThread.run()	+34 library calls	242.44 kB
< 1%	jdk.internal.misc.InnocuousThread.run()	+4 library calls	472 B
< 1%	java.lang.Thread.<init>(ThreadGroup, Runnable)	+3 library calls	424 B

pooled: 1.31GB



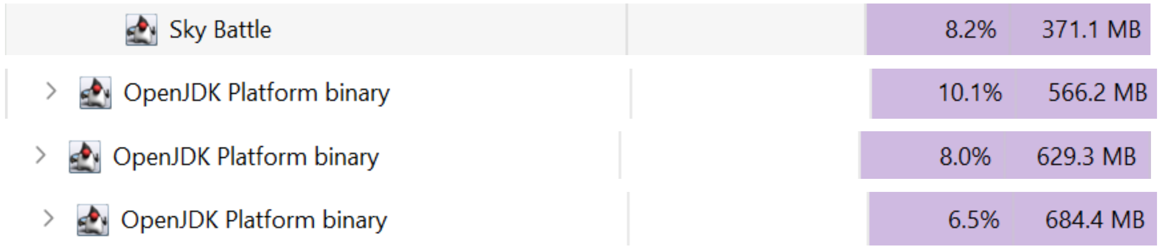
Method			Allocation size
98.7%	java.lang.Thread.run()	+1,864 library calls	1.31 GB
1.1%	java.lang.Thread.exit()	+4 library calls	14.19 MB
< 1%	com.sun.glass.ui.InvokeLaterDispatcher.run()	+6 library calls	1.75 MB
< 1%	java.util.TimerThread.run()	+30 library calls	610.95 kB
< 1%	com.sun.javafx.tk.quantum.QuantumToolkit.lambda$runToolkit$12()	+11 library calls	488.81 kB
< 1%	com.sun.media.jfxmediaimpl.NativeMediaPlayer$EventQueueThread.run()	+17 library calls	170.2 kB
< 1%	jdk.internal.misc.InnocuousThread.run()	+4 library calls	472 B
< 1%	java.lang.Thread.<init>(ThreadGroup, Runnable)	+1 library call	368 B

D) Tracking memory fluctuation after 1, 3, 6, 8 game sessions in a single application launch

One session: at least played through Level 1 and Level 2

nopool:

- 371MB after first session
- 566MB after third session
- 629MB after sixth session
- 684MB after eighth session

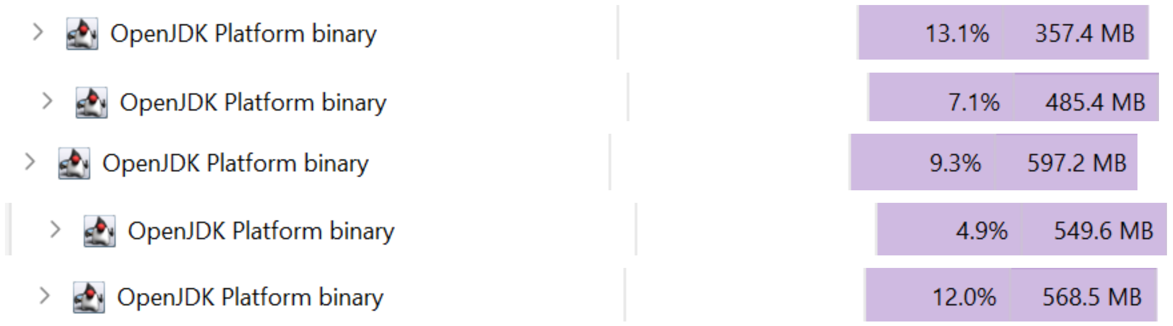


Sky Battle	8.2%	371.1 MB
OpenJDK Platform binary	10.1%	566.2 MB
OpenJDK Platform binary	8.0%	629.3 MB
OpenJDK Platform binary	6.5%	684.4 MB

pooled:

**interestingly memory usage actually decreased after sixth session, unsure about mechanics behind it*

- 357MB after first session
- 485MB after third session
- 597MB after sixth session
- 550MB after eighth session
- 568MB after ninth session



OpenJDK Platform binary	13.1%	357.4 MB
OpenJDK Platform binary	7.1%	485.4 MB
OpenJDK Platform binary	9.3%	597.2 MB
OpenJDK Platform binary	4.9%	549.6 MB
OpenJDK Platform binary	12.0%	568.5 MB