FACULTY OF ENGINEERING AND ARCHITECTURE

DEPARTMENT INTEC
RESEARCH GROUP IBCN

Geavanceerde softwareontwikkeling

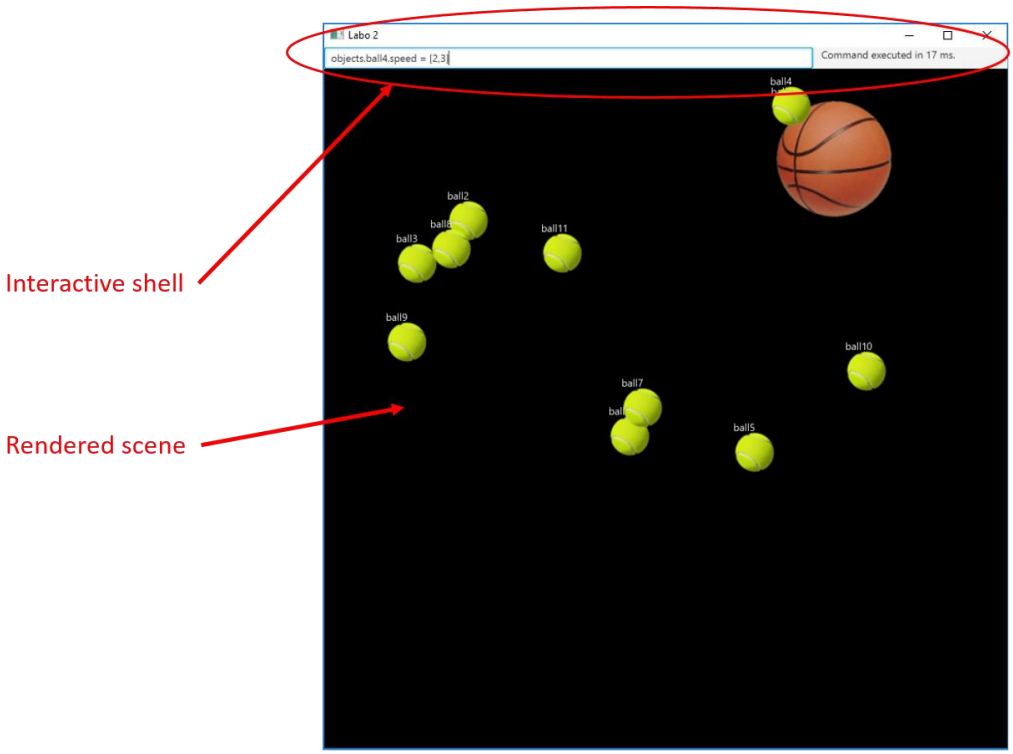
Lab Session II

Singleton, Object Pool and Flyweight

GHENT UNIVERSITY

Introduction

In this Lab we'll show you how design patterns such as Object Pool and Flyweight can really make a difference in applications where performance matters. We've setup a demo application that implements a basic game engine and attached an interactive shell that allows you to interact with the rendered scene (spawn new objects, attach movement vectors, etc.).

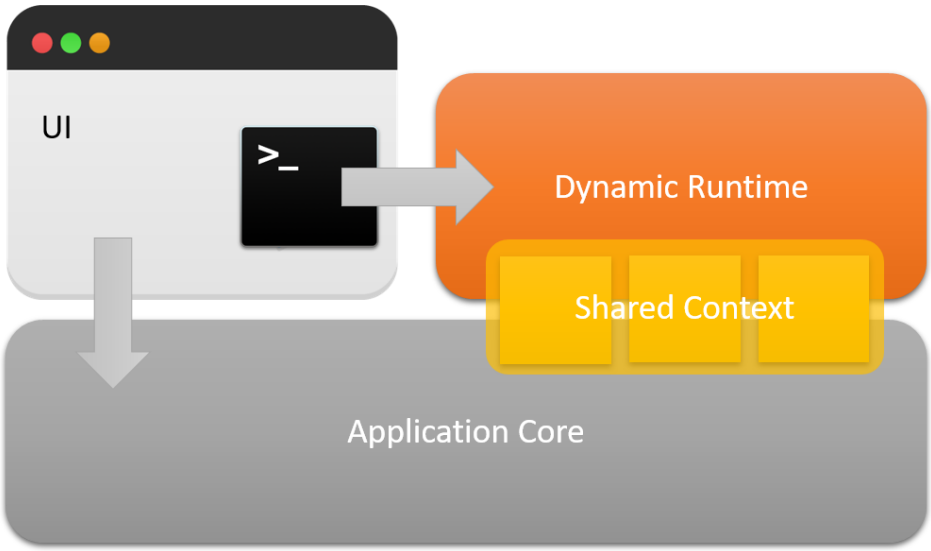


When using the application you'll quickly notice that the commands issued using the shell take some time to execute and that the animations start to become less fluent once there are a lot of objects on the screen.

It is your task to optimize the application by applying the Object Pool and Flyweight design patterns to the right parts of the game engine.

Interactive Shell

Before introducing the optimizations, we'll first give some more information on the interactive shell and how it was implemented. Applications that expose an interactive shell (games with developer consoles, database interfaces, etc.) will generally be designed as follows:



The user interacts with the application through the UI and for each action that is performed (a button is clicked or keys are typed), a function is called in the compiled Application Core. When the interactive shell is activated, the user can start typing commands with various parameters. This is clearly not as straightforward as calling a function for a fixed UI action.

The first step is parsing the command issued by the user. The application will have a well-defined set of available commands and some syntax rules concerning the language that is used to express the commands. Once the command is parsed and deemed syntactically correct, it can be executed by a Dynamic Runtime, which is the component of the application that can translate dynamic shell commands into the functions that need to be called in the Application Core. To do this, the Dynamic Runtime will make use of a number of context objects that are shared with the Application Core (this is the API of the application exposed to the shell).

In our demo application, we use JavaScript as the dynamic language of choice. The JavaScript Engine **Nashorn**, which is embedded in the Java system, provides us with a Dynamic Runtime to execute the commands issued as JavaScript statements. The available commands are declared as JavaScript functions in a script-file that is loaded each time the runtime is started (*see commands.js*). Our Application Core (the game engine) shares the collection of scene objects with the runtime so the items rendered on screen can be manipulated using the available commands.

The following table lists the JavaScript commands that are available for the shell:

createTennisBall	Typing `createTennisBall()` spawns a tennis ball on a random location in the scene. You can specify an optional location by providing x and y coordinates as arguments, e.g. `createTennisBall(300,300)`
createFootBall	Similar to *createTennisBall*, but spawns a football instead.
createBasketBall	Similar to *createTennisBall*, but spawns a basketball instead.
repeat	Repeats a specific command for the indicated number of times, e.g. say you want to spawn 30 tennis balls, then you can type: `repeat(30, createTennisBall)` Notice the lack of parentheses after createTennisBall: the function is not being called here, but given as an argument to the repeat function.
deleteAll	Typing `deleteAll()` removes all objects from the scene.
moveAll	Typing `moveAll()` assigns a random movement vector to all objects in the scene.

In addition to the above commands, the shell also provides access to the shared objects collection that holds all the scene objects. We can use this to manipulate the objects on the screen:

Example 1, changing the location of ball1

```
objects.ball1.location = [50,150]
```

Example 2, setting the speed of ball1:

```
objects.ball1.speed = [15,15]
```

Example 3, changing the type of ball1:

```
objects.ball1.type = ObjectType.FOOTBALL
objects.ball1.type = ObjectType.BASKETBALL
objects.ball1.type = ObjectType.TENNISBALL
```

Play around with these commands a bit to become familiar with the behavior of the application and take a look at the provided source files to get an understanding of how the interactive shell was implemented.

Optimizing the Application

Shell execution time (Object Pool)

To prevent the user from breaking the interactive shell (by overriding certain commands or reassigning the *objects* variable) a new instance of the runtime is created for each command that is executed. The drawback of this approach is that the commands take a relatively long time to execute because of the initialization overhead of the runtime.

To optimize this aspect of the application, you will design and implement an object pool based on the supplied ObjectPool<T> interface that pre-initializes a fixed number **(e.g. 4 should be more than enough)** of CommandRuntime instances, so our game engine can immediately execute the input command by retrieving an already initialized instance from the pool. Start by creating a new class CommandRuntimePool that implements ObjectPool<CommandRuntime>. Notice that we instantiate the generic parameter T of ObjectPool with the type CommandRuntime. This means that CommandRuntimePool will be a specific implementation of ObjectPool that only provides CommandRuntime instances.

To complete the code for CommandRuntimePool, you will need to implement three methods:

1. A constructor that takes on an instance of Engine as parameter. You will need a reference to this Engine to create new CommandRuntime objects. Use some kind of collection to store the fixed number of pre-initialized CommandRuntime instances.
2. The getInstance method that returns a CommandRuntime instance that is ready (i.e. the instance is newly initialized or has been reset, use the isReady() method to check this). If no instance can be found that is ready, you may throw a RuntimeException.
3. The releaseInstance method that takes as parameter a CommandRuntime instance and calls the reset() method on it, so the instance can be made ready for re-use in the pool.

Important: Take a good look at how the reset() method is implemented in CommandRuntime. The initialization steps (called by reset) can take some time to complete and we don't want our program to wait for that every time we release a CommandRuntime instance. That is why the reset method will call the re-initialization in a separate thread (a lightweight process) that will allow the initialization code to be executed in parallel. Running tasks in parallel is called concurrent computing and for more information on this topic we refer to the theory classes and the excellent article that can be found here: http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

Changes to the Engine

To apply the shell execution time optimization, modify the Engine.java class in the following way:

1. Initialize the Object Pool in the Engine constructor
2. Write code in the `processInput` method to allow executing commands using the Object Pool.
3. Set the constant `OPTIMIZE_SHELL_EXECUTION_TIME` to `true`.

Animation Lag (Flyweight and Singleton)

Each scene object has a type attribute that is used in the render loop to determine which sprite to draw on screen. The scene is rendered about 30 times per second, so a lot of new Sprite objects (JavaFX Image objects) are continuously created and loaded from disk. As a result, the animations start to stutter once there are a lot of objects on screen.

To optimize this aspect of the application, you will design and implement a flyweight class that manages the creation of the sprites (SpriteLoader.java). By using the flyweight pattern, we can guarantee that only one instance is created for each of the sprites that is used in the application. As a result, a lot less Image instances need to be allocated while the application is running and the sprites are now stored in memory, which leads to less overhead when drawing the sprites in the render loop.

Implement the SpriteLoader class as a singleton so it can easily be accessed from the Engine.

Changes to the Engine

To apply the animation optimization, modify the Engine.java class in the following way:

1. Write code in the `getSprite` method to allow retrieving Sprites using the `SpriteLoader` singleton instance.
2. Set the constant `OPTIMIZE_ANIMATIONS` to `true`.

Submitting your solution

1. Important: labs are solved individually
2. Bundle your source solution files in one zip file named ***lab2_name_surname.zip*** (e.g. *lab2_bruno_volckaert.zip*)
3. The following files need to be included in the zip file:
    a. Engine.java
    b. CommandRuntimePool.java
    c. SpriteLoader.java
4. Send this zip-file via Minerva dropbox to **Wannes Kerckhove, Thomas Dupont** and **Bruno Volckaert**, deadline for sending this is exactly one week from the start of this lab. I.e. for this lab the deadline is Friday 20th of October at 15:59.