Aurora Tiffany-Davis
(attiffan)

TABLE OF CONTENTS

Part 1 - Scripting **1**

Part 2 - A Second Game **2**

Part 3 - A Third Game **3**

Part 4 - Reflection **4**

Screenshots **7**

Startup Terminal Message 7

Ongoing Play 8

Pause 8

1. Part 1 - Scripting

1.1. Scripting Language

Javascript was chosen as the scripting language for several reasons. First, the provided demonstration code used this language. Second, I am very familiar with this language. Third, this is a very popular language, so this choice could make the game engine more supportable (by other developers) in the future.

1.2. Script Manager

The Script Manager provides a modest collection of features which were more than sufficient to support the current needs of the game engine. The Manager provides the ability to load a script. The Manager provides the ability to invoke a script function. Currently there are two signatures of the *invokeFunction()* method - more could be added if needed in the future. The first signature allows the game engine to invoke a method that does not take arguments, but that does return a value. The second signature allows the game engine to invoke a method that does take arguments, but that does not return a value. The Manager also provides the ability to bind an argument, although the current game engine does not make use of this ability.

1.3. Scripts

1.3.1. Game-Specific Scripts

There is one script for each supported game. Each script has the same methods, so that the game engine may call these methods at appropriate times, in a game-agnostic way. This approach was chosen in order to maximize the flexibility of the game engine. A few of the methods are discussed below, to give an idea of how these scripts provide flexibility.

Method	Executed	Example Behavior
performGameLoopIterationServer	Each regular update cycle (game loop iteration) of the Server	Platformer moves platforms around. Bubble Shooter moves bubbles and shooting arrow around.
handleEventUserInput	When the Client character object receives a user input event for which it has registered	Platformer updates character attributes to set jump and left-right motion. Bubble Shooter updates arrow attributes to set angle, or shoots bubble.

Other game script methods: *instructUser(), getSpaceBarAction(), isReplayEnabled(), setUpNewCharacter(), populateGameWorld(), performGameLoopIterationClient()*

1.3.2. Game-End Script

In order to exercise the Script Manager fully, a game-end script was also added. This exercises the ability to change which script is loaded, on the fly. This game-end script is loaded, and a method within it executed, when the game ends. The script provides feedback on the Client terminal regarding win / loss, and number of points earned.

2. Part 2 - A Second Game

2.1. Bubble Shooter

This game was selected because it introduces multiple new game objects (bubbles and shooting arrow), and multiple new game dynamics (addition of game objects mid-game, and the concept of shooting).

2.2. Feature Support

Pause is supported, but replay and multi-player modes are not supported.

2.3. Implementation

2.3.1. General Implementation

As noted in "Part 1 - Scripting", there is one script for each supported game. Each script has the same methods, so that the game engine may call these methods at appropriate times, in a game-agnostic way. This approach was chosen in order to maximize the flexibility of the game engine.

Game objects needed by Bubble Shooter are implemented in native code rather than in the game-specific script. This is done in case some future game may also need such game objects as bubbles or shooting arrows.

2.3.2. Bubble Objects

Bubble objects have a *shootBubble()* method so that the game engine can respond immediately to user input. The bubble location is then updated on screen during each regular update cycle (game loop iteration). This is to keep a separation of concerns, as a good practice for maintainability. Like the platforms in the first game (Platformer), the speed at which a bubble moves is frame rate independent (making use of timeline classes).

Bubble objects have a *bShooterBubble* attribute to support the difference between a shooter bubble colliding with something (may pop a bubble) and a non-shooter bubble colliding with something (does not pop anything). This is also used to initiate a chain reaction of collision events when a shooter bubble collides with a non-shooter bubble of the same color. Any such non-shooter bubbles are changed into shooter bubbles so that other non-shooter bubbles of the same with which they are already collided may be popped as well. This approach was used to take advantage of the existing event functionality in the game engine (rather than writing extra code to, for example, perform a breadth-first search to find adjacent bubbles of the same color).

The bubble object class has a *addNewRowOfBubblesAtTopOfWindow()* method so that game world population can easily change the number of initial rows of bubbles, and so that advancing bubbles during gameplay can also be easily supported.

2.3.3. Arrow Objects

Arrow objects have an *updateAngle()* method so that the angle of an arrow can change immediately based upon user input. The angle is then updated on screen during each regular update cycle (game loop iteration). This is to keep a separation of concerns, as a good practice for maintainability. Like the platforms in the first game (Platformer), the speed at which the arrow's attributes change in response to user input is frame rate independent (making use of timeline classes).

Arrow angles are clamped between -90° and 90°.

2.4. Reuse

2.4.1. Quantitative Comparison

The first game (Platformer, after scripting was added) is compared with the second game (Bubble Shooter), using *git diff --shortstat [commit1] [commit2]*. Results: *35 files changed, 2498 insertions(+), 874 deletions(-)*

With the scripting examples and the Processing core.jar file deleted, the first game has 11209 lines of code (per cloc v1.74). This includes comment lines (I am a prolific commenter) and blank lines.

(2498 + 874) / 11209 ≈ 30% change.

However, much of this change was due to adding classes for new Bubble Shooter game objects, as well as filling in the *HW04_Game_BubbleShooter.js* script file. Also, I am in the habit of constantly fiddling with comments and spacing while organizing my thoughts.

2.4.2. Qualitative Comparison

It may be more instructive to discuss the most meaningful differences between these commits. This is done in "Part 4 - Reflection".

3. Part 3 - A Third Game

3.1. Space Invaders

This game was selected for ease, due to its similarity to Bubble Shooter. Enemies are analogous to non-shooter bubbles, and bullets are analogous to shooter bubbles.

3.2. Feature Support

Pause is supported, but replay and multi-player modes are not supported.

3.3. Implementation

3.3.1. General Implementation

As noted in "Part 1 - Scripting", there is one script for each supported game. Each script has the same methods, so that the game engine may call these methods at appropriate times, in a game-agnostic way. This approach was chosen in order to maximize the flexibility of the game engine.

Game objects needed by Space Invaders are implemented in native code rather than in the game-specific script. This is done in case some future game may also need such game objects as enemies or bullets.

3.3.2. Enemy Objects

The enemy object class has a *moveEnemies()* method which moves enemies and also (based on random timing) fires bullets from them.

The enemy object class has a *destroyEnemy()* method which can be called from a bullet object's *handleCollision()* method.

The enemy object class has a *addNewRowOfEnemiesAtTopOfWindow()* method so that game world population can easily change the number of initial rows of enemies. This method calls the *moveAllEnemiesDown()* method, so that advancing enemies during gameplay can also be easily supported.

3.3.3. Bullet Objects

Bullet objects have a *handleCollision()* method which handles the difference between a bullet striking an enemy (player scores a point, enemy is destroyed), striking a character (player loses the game), and striking another bullet (both bullets are destroyed). Regardless of what a bullet strikes, it is removed from the game. If a character shot this bullet, then the character is again allowed to shoot (characters cannot shoot indiscriminately).

The bullet object class has a *shootBullet()* method which takes as an argument another game object, to support shooting a bullet up from a character object, or down from an enemy object.

3.4. Reuse

3.4.1. Quantitative Comparison

The second game (Bubble Shooter) is compared with the third game (Space Invaders). Results: *30 files changed, 1389 insertions(+), 140 deletions(-)*

As noted previously, the first game has 11209 lines of code (per cloc v1.74). This includes comment lines and blank lines.

(1389 + 140) / 11209 ≈ 14% change.

However, much of this change was due to adding classes for new Space Invaders game objects, as well as filling in the *HW04_Game_SpaceInvaders.js* script file. Also, I am in the habit of constantly fiddling with comments and spacing while organizing my thoughts.

3.4.2. Qualitative Comparison

It may be more instructive to discuss the most meaningful differences between these commits. This is done in "Part 4 - Reflection".

4. Part 4 - Reflection

4.1. Changes to Game Engine to Support Second Game (Bubble Shooter)

4.1.1. HW04_Client.java

The character object now registers for SCORE_CHANGE events globally rather than locally, since it is now possible for the Server to change a character's score.

Some keypress code is wrapped in an IF statement and only runs if replay mode is enabled, since it is now possible for a game to not support replay mode.

4.1.2. HW04_Color.java

The methods *getRandomColor()* and *getPureColor()* are added to support bubbles of various colors.

4.1.3. HW04_EventManager.java

Some code in the *raiseEventScoreChange()* method is wrapped in an IF statement to support a SCORE_CHANGE event taking place in a single-player-only game.

4.1.4. HW04_Globals.java

The variable oMyCharacterObject is made available globally as a convenience for game-specific scripts.

4.1.5. HW04_ObjectBoundary.java

A boundary-related method is moved into this file from *HW04_Server.java*, as adding more such methods to support multiple games was "junking up" *HW04_Server.java*.

4.1.6. HW04_ObjectCollidable.java

The method *checkForCollisionAndRaiseEvent()* is added to avoid repeated code, as this repeated code was proliferating with the addition of a new game.

The interpretation of what constitutes a collision is broadened to include objects that are merely touching, to support multi-bubble popping in Bubble Shooter.

4.1.7. HW04_ObjectCommunicable.java

The method *castObject()* is expanded to work also with new game objects used by Bubble Shooter.

4.1.8. HW04_ObjectMoveable.java

The method *updateLocationAndCheckForCollisions()* is added to avoid repeated code, as this repeated code was proliferating with the addition of a new game.

4.1.9. HW04_ObjectPlatform.java

Several platform-related methods are moved into this file from *HW04_Server.java*, as adding more such methods to support multiple games was "junking up" *HW04_Server.java*.

4.1.10. HW04_ObjectSpawnPoint.java

A spawn-point-related method is moved into this file from *HW04_Server.java*, as adding more such methods to support multiple games was "junking up" *HW04_Server.java*.

The method *addSpawnPointToBottomOfWindow()* is added to support this feature of Bubble Shooter (which could also apply to other future games).

4.1.11. HW04_ObjectStatusInstructions.java

The logic for determining what instructions to display is made more flexible, with regard to support (or non-support) for replay mode, and with regard to what the spacebar input does in a game. In both cases, the information needed for the logic is returned from game-specific script methods.

4.1.12. HW04_ObjectZoneDeath.java

A death-zone-related method is moved into this file from *HW04_Server.java*, as adding more such methods to support multiple games was "junking up" *HW04_Server.java*.

4.1.13. HW04_ObjectZoneWin.java

A win-zone-related method is moved into this file from *HW04_Server.java*, as adding more such methods to support multiple games was "junking up" *HW04_Server.java*.

4.1.14. HW04_Script_Platformer.js

The method *isReplayEnabled()* is added because games now may or may not enable replay mode.

Some code in the *handleEventUserInput()* method is wrapped in an IF statement to support this method being called by the game engine on either the Server or the Client.

4.1.15. HW04_Server.java

The server now registers for USER_INPUT events.

The method *removeObjectFromGame()* is added to avoid repeated code, as this repeated code was proliferating with the addition of a new game.

4.2. Changes to Game Engine to Support Third Game (Space Invaders)

4.2.1. HW04_EventManager.java

The method *deRegisterForEvents()* is added to aid in garbage collection when game objects are no longer needed, for example when popping bubbles or when bullets leave the screen.

4.2.2. HW04_NetworkPartnerProxy.java

Some code in the *handleEvent()* method is wrapped in an IF statement to avoid a network partner sending an event to itself, an edge case not seen until Space Invaders was implemented.

4.2.3. HW04_ObjectBubble.java

Several methods are added to support deferred popping of a bubble, which is needed to work with deregistered events on popped bubbles in a bubble-popping chain reaction. In other words, deregistration was added for Space Invaders, but "broke" something in Bubble Shooter.

4.2.4. HW04_ObjectCharacter.java

Several methods are added to support a character being allowed (or not allowed) to shoot a bullet, to support the Space Invaders game mechanic of only being allowed to shoot a second bullet after your first has hit something or left the screen.

4.2.5. HW04_ObjectCommunicable.java

The method *castObject()* is expanded to work also with new game objects used by Space Invaders.

4.2.6. HW04_ObjectEvent.java

The priority of the SCORE_CHANGE event is increased to support changing the score and ending the game in quick succession.

4.3. Reuse Thoughts

There were many aspects of the game engine for which flexibility was not originally considered. The exercise of making changes to support additional games was very valuable in making the game engine more capable than it was before. It is expected that with each additional game, more changes in support of flexibility will be needed, but that those changes will decrease with each additional game. So far this trend holds, as more lines of code changed with the addition of the second game, than with the addition of the third game.

4.4. Lessons Learned

4.4.1. Things I Would Change If I Did It Again

4.4.1.1. Monolithic Game Object Hierarchy

The game engine uses a monolithic game object hierarchy, which is quite inflexible. I faced issues of classes inheriting capabilities that were inappropriate or not useful. The most troubling aspect of this was the *HW04_ObjectArrow* class, added for the Space Invaders game. Arrows inherit from *HW04_ObjectRenderable* so that they may be displayed on screen. As a consequence, they also inherit from *HW04_ObjectCollidable*, even though in practice collisions are meaningless for arrows. The *HW04_ObjectArrow* class has a no-op *handleCollision()* method, and arrows will be found and reported as colliding objects, causing inefficiencies in other *handleCollision()* methods that iterate through colliding objects.

4.4.1.2. Globals

The *HW04_Globals* class exists as a convenience, to expose objects and attributes that are very widely used in the engine. However, these objects and attributes properly belong in classes, exposed via explicit "getter" methods.

4.4.1.3. Accessing All Objects of a Given Type

Many game objects class have a method like *getCollidableObjects()* or *getSpawnPointObjects()*. It would have been better to take the time to figure out how to return all objects of a given type in a more generalized way, to avoid repeated very similar code.

4.4.1.4. Event Attributes as Strings

Each event object has an attribute *oArgs*, which is a hashmap with a String as the key and an Object as the value. Using Strings for keys was very flexible, but also very error-prone, given the likelihood of a developer committing a simple spelling mistake. I would revisit this and would consider using an enum instead.

4.4.1.5. Casting of Game Objects

The *HW04_ObjectCommunicable* class has a method *castObject()*, which is used to cast objects to a specific game object class so they can be worked with more easily. However, this method uses a big switch statement, switching on the *sObjectType* attribute that all communicable objects have. As a result, a new case must be added to the switch statement with every new game object type. I would revisit this and take the time to figure out how to do this in a more generalized way.

4.4.2. Things I Would Keep the Same If I Did It Again

4.4.2.1. Single Event Object Class

The use of a single class for event objects proved to be a big advantage for flexibility, as adding new classes and changing of event priorities took place throughout the evolution of the game engine. There was a significant amount of logic that applied regardless of event type.

4.4.2.2. Network Partner Proxies

An object may register interest in an event type from other network partners. In this case, the event manager may find a network partner proxy object in the collection of objects that is interested in an event. To the event manager, this is just another interested object. The network partner proxy object will handle the event by communicating it to the appropriate network partner. This proved to be a useful bit of code that did not have to change as additional events were added, as objects changed whether they were interested in certain event locally vs. globally, and as new games were added.

4.4.2.3. Communicable Object Class

Each game object, and each event object, inherits from this class, which has two attributes assigned at object creation: *sObjectType* and *nPlayerID*. It proved very useful to have these two pieces of information firmly established from the point of creation for every single game object and event. This information was useful in casting game objects to take advantage of their class's capabilities, and in handling game objects and events based upon where they originated.

4.4.2.4. Blocking .take() for Communicated Objects

The Client and Server do not use .wait() and .notify() to perform writes to network partners. Instead, the write thread simply uses .take(), a blocking dequeue operation, to get the next object to write out. This is the same strategy used to pull events from the event queue in the event manager and in the logger. In my experience, this approach resulted in communication and synchronization that was flexible and easy to understand.

4.4.2.5. Randomization

The game engine uses randomization in many areas, including but not limited to platform size and placement in the Platformer game, bubble coloring in the Bubble Shooter game, and enemy firing patterns in the Space Invaders game. This randomization was primarily intended to improve gameplay, however it has an additional advantage, which is that it aids in testing and debugging - it helps to find edge cases that the developer might not have thought of.

4.4.2.6. Logging

Both the Client and the Server log (in separate files) all events of which they are made aware, with timestamps. Although this feature was not tested for the two new games added in this assignment, in general this has been a useful feature for debugging and sanity checking.

4.4.2.7. Error Handling

The game engine incorporates some simple error handling. Almost every function has a try/catch block, with any exceptions being handed off to a centralized error handling function. The same error handler is used on both the server and client, to provide a consistent look and feel for the users and for developers. At this still early stage of game engine development, any error causes the program to terminate immediately, to speed up debugging.

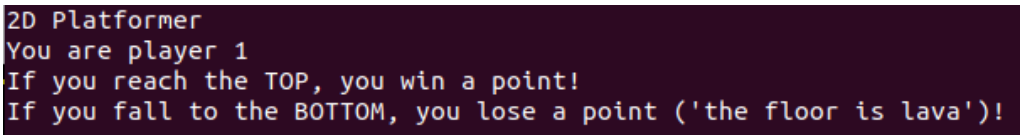
4.4.2.8. Code Comments

I am in the habit of commenting prolifically while writing code, usually filling in comments even before writing a new bit of code. This proved very useful. Because the game engine is relatively complex, because it was developed over the course of months, and because I have other demands on my memory, looking at old code was often like looking at code someone else had written. The comments I made were a gift to my future self.

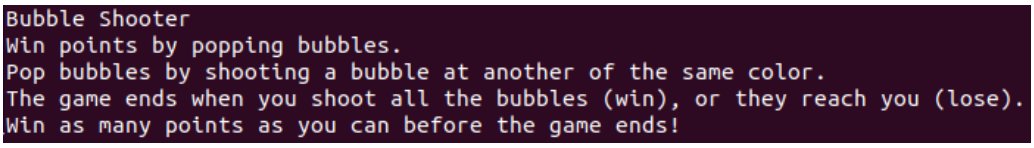
5. Screenshots

5.1. Startup Terminal Message

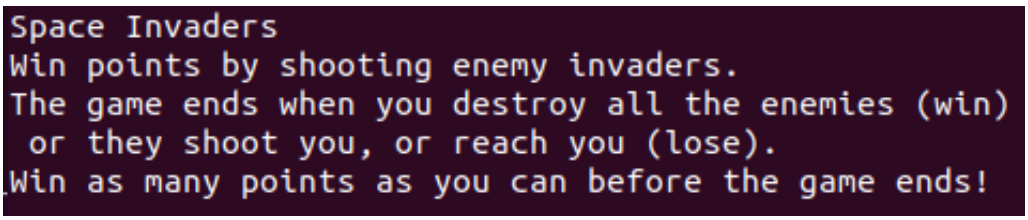
Platformer



Bubble Shooter

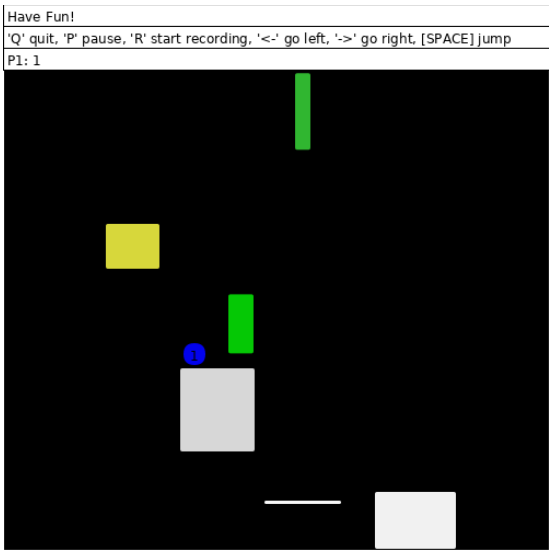


Space Invaders

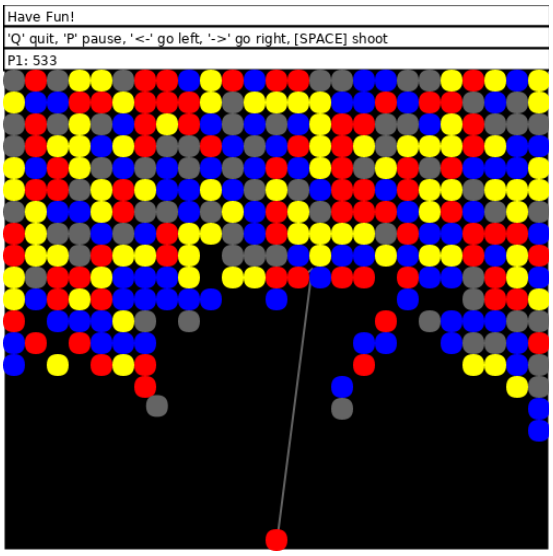


5.2. Ongoing Play

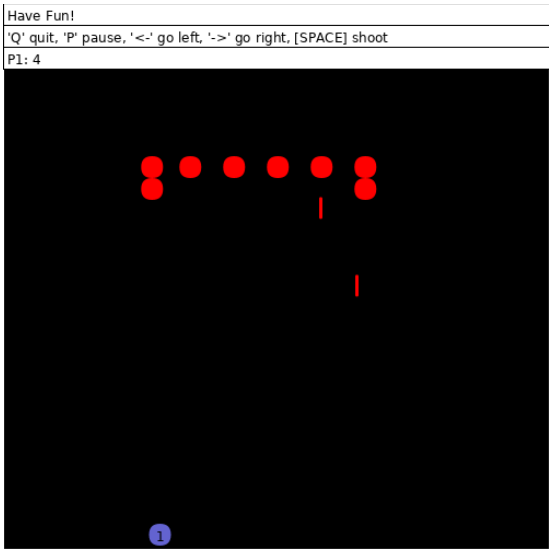
Platformer



Bubble Shooter

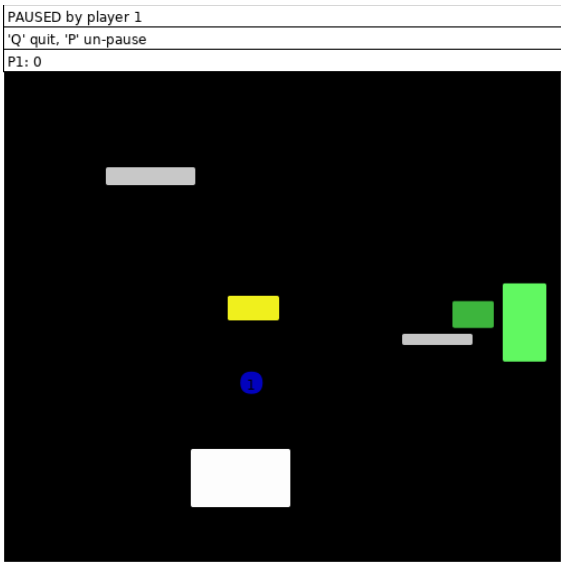


Space Invaders

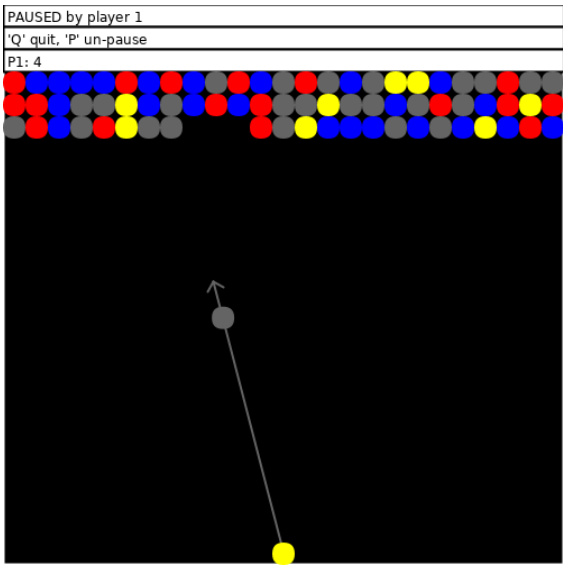


5.3. Pause

Platformer



Bubble Shooter



Space Invaders