IST400/600 Scripting for Games, Spring 2010
Lab 11
Instructor: Keisuke Inoue

General Description:

In this lab, you will create components of a scrolling game that were discussed in the previous class. The following instructions will help you creating them, but will not tell you all the small steps. The lab is designed to encourage your learning and enhance your understanding by working independently. **You may ask questions to instructors and/or friends (but keep it quiet) and look up previous lecture materials, textbook, and other resources. Do not copy-and-paste any resources other than you created.**

Implementation Outline:

Work on one (or both, if you would like) of the following tasks:

1. **"Tokens", that the main character can collect and gain scores.**
2. **"Enemies" and "missiles" that the main character can shoot and destroy the enemies.**

This lab is shorter than usual, so that you will have time to improve your game for the final project, e.g. working on the graphics and other details, designing and implementing your original features, etc.

Lab Instructions:

0. Preparations

This lab will be done on top of what you have created in Lab10. You should have submitted your Lab10 before you submit this one. Open your "Lab10.html" with your text editor, and save it as a new it with a name "Lab11.html" (or whatever you like).

You will need graphic files for tokens, missiles, enemy characters, or explosions depending on the task you would like to do.

Task 1. Implementing tokens

In this task, you will improve the game by implementing "tokens", objects that the main character can collect and gain scores during the game. They are called "tokens" in this document, but they could be anything: coins, treasures, fruits, etc. The idea is once the character and the token are collided, the character should gain scores.

As discussed in the class, the implementing tokens is very similar implementing obstacles, so this is your chance to show-off your copy-and-paste skills. There are five steps:

1) Modifying `initializeGame()` to set up necessary global variable properties.
2) Modifying `clearGame()` to set up some other necessary global variable properties.
3) Defining `createToken()`, which creates a token and display it.
4) Defining `updateTokens()`, which updates the states (position, etc,) of tokens.
5) Defining `clearTokens()`, which removes all the tokens in the display.
6) Modifying `updateGame()` to call `createToken()` and `updateTokens()` as needed.

1.1) Modifying initializeGame()

In this function, initialize the following properties:

1) `gGameState.score`, which keeps track of the score of the score. This property should be zero at the beginning of the game.

2) `gGameState.tokenNum`, which keeps track of the number of tokens that have been generated (for generating the IDs of the HTML elements) and should be zero at the beginning of the game.

1.2) Modifying clearGame()

In this function, initialize the following properties. All the properties are familiar to you, since you have used the same properties for obstacles.

1) `gGameState.tokenList`
2) `gGameState.tokenSpeed`
3) `gGameState.tokenInterval`
4) `gGameState.tokenStartingLeft`
5) `gGameState.tokenTopLimit`
6) `gGameState.tokenBottomLimit`

1.3) Defining createToken ()

The definition of this function is almost same as the definition of `createObstacle()`. Below are a few points that you should modify:

1) You need to use the token-related properties (all declared above), instead of the obstacle-related properties. (Basically, you should not see `gGameState.obsXXX` in this function!)
2) Make sure to change the prefix for the `id` attribute to something else, so that JavaScript won't get confused with obstacles and tokens.
3) Make sure to use a different graphic file.

1.4) Defining updateTokens ()

Again, the definition of this function is almost same as the definition of `updateObstacles()`. Below is the point that you need to modify, other than things that were listed above:

1) When the collision is detected, you only need to do two things:

    1) Increase `gGameState.score` (by whatever the amount you like).
    2) Remove the token from the document body.

1.5) Defining clearTokens ()

The definition of this function is exactly the same as the definition of `clearObstacles()`, except the modifications already mentioned. Make sure this function is called when it is needed. (Where is that?)

1.6) Modifying updateGame ()

Insert function calls to execute `createToken()` and `updateTokens()`, just like `createObstacles()` and `updateObstacles()`. You may want to modify the condtions for the if statement or the values of `gGameState.tokenInterval` and `gGameState.obsInterval`.

At this point, save the file and reload it on Firefox. Start the game and check if the tokens are displayed properly.

Task 2. Implementing "enemies" and "missiles".

In this task, you will modify the game so that the character can shoot some kind of missiles to shoot down enemy characters. (Again, “missiles” don’t have to be missiles and “enemies” don’t have to be enemies. Be creative and come up with your own design!)

This task is divided into two subtasks:
2.1) Implementing enemies and
2.2) Implementing missiles.

If you want to make the game so that you can shoot down the obstacles, you can skip the subtask 2.1).
(But remember that I will keep using the word “enemy”, which will mean “obstacles” to you.)

Subtask 2.1. Implementing enemies

Just like tokens, implementing enemies is very similar to implementing obstacles, so this is your chance to show-off your copy-and-paste skills. (And yes, I clearly copied and pasted the instructions from above. ☺)

1) Modifying `initializeGame()` to set up the necessary global variable properties.
2) Modifying `clearGame()` to set up the necessary global variable properties.
3) Define `createEnemy()`, which creates a token and display it.
4) Define `updateEnemies()`, which updates the states (position, etc,) of tokens.
5) Define `clearEnemies()`, which removes all the tokens in the display.
6) Modifying `updateGame()` to call `createEnemy()` and `updateEnemies ()` as needed.

2.1.1) Modifying initializeGame()

In this function, initialize the following property:

1) `gGameState.enemyNum`, which keeps track of the number of enemies that have been generated (for generating the IDs of the HTML elements) and should be zero at the beginning of the game.

2.1.2) Modifying clearGame()

In this function, initialize the following properties. All the properties are familiar to you, since you have used the same properties for obstacles.

1) `gGameState.enemyList`
2) `gGameState.enemySpeed`
3) `gGameState.enemyInterval`
4) `gGameState.enemyStartingLeft`
5) `gGameState.enemyTopLimit`
6) `gGameState.enemyBottomLimit`

2.1.3) Defining createEnemy ()

The definition of this function is almost exactly the same as the definition of `createObstacle()`. Below are a few points that you should modify:

1) You need to use the enemy-related properties (all declared above), instead of the obstacle-related properties. (Basically, you should not see `gGameState.obsXXX` in this function!)
2) Make sure to change the prefix for the `id` attribute to something else, so that JavaScript won’t get confused with obstacles and tokens.
3) Make sure to use a different graphic file.

2.1.4) Defining updateEnemies ()

The definition of this function is exactly the same as the definition of `updateObstacles()`, except the modifications already mentioned.

The handling of the collision is up to you, but the same definition as the `updateObstacles()` (i.e., If you collide with an enemy, you are dead.) works fine.

2.1.5) Defining clearEnemies ()

The definition of this function is exactly the same as the definition of `clearEnemies()` (except the modifications already mentioned.) Make sure this function is called when it is needed. (Where is that?)

2.1.6) Modifying updateGame ()

Insert function calls to execute `createEnemy()` and `updateEnemies()`, just like `createObstacles()` and `updateObstacles()`. You may want to modify the condition for the if statement or the values of `gGameState.enemeyInterval` and `gGameState.obsInterval` in order to make the obstacles and/or enemies to appear in good mixture.

At this point, save the file and reload it on Firefox. Start the game. You should see the enemy characters appear in the game, just like obstacles.

Subtask 2.2. Implementing missiles

Once again, implementing missiles is very similar to implementing obstacles. Below are the steps. The step 3) is new, because you want to control shooting of missiles from a keyboard, but not by a certain intervals.

1) Modifying `initializeGame()` to set up the necessary global variable properties.
2) Modifying `clearGame()` to set up the necessary global variable properties.
3) Define `createMissile()`, which creates a token and display it.
4) Modifying `moveRover()` to call `createMissile()` as needed.
5) Modifying `updateGame()` to call `updateMissiles()` as needed.
6) Define `updateMissiles()`, which update the states (position, etc,) of tokens.
7) Define `clearMissiles()`, which remove all the tokens in the display.

2.2.1) Modifying initializeGame()

Initialize the following property:

1) `gGameState.missileNum`, which keeps track of the number of missiles that have been generated (for generating the IDs of the HTML elements) and should be zero at the beginning of the game.

2.2.2) Modifying clearGame()

In this function, initialize the following properties. All the properties are familiar to you, since you have used the same properties for obstacles.

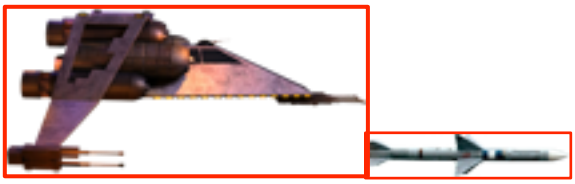
1) `gGameState.missileList`
2) `gGameState.missileSpeed`

They are much fewer than in the case of implementing enemies or obstacles, because of the definition of `createMissile()` is different from the definition of `createEnemy()` or `createObstacle()`, which is explained below.

2.2.3) Defining createMissile ()

The definition of this function is somewhat different from the definition of `createObstacle()` (other than the points that have been mentioned already), because you need to initialize the starting position of a missile relative to the current position of the main character.

For example, the following shows an example of starting position of a missile, relative to the position of a main character.



Below is an example of how the initialization can be done to achieve positioning like above:

```
var sp = document.body.getElementById('sp_main');

missile.style.left = parseInt(sp.style.left) + sp.width;
missile.style.top = parseInt(sp.style.top) + sp.height - missile.height;
```

You should come up with your own definition of the coordinates to suit your need.

2.2.4) Modifying moveRover ()

Unlike other objects that you have implemented so far (obstacles, tokens, or enemies), you may not want to call `createMissile()` from `updateGame()`, because it does not make much sense to shoot missiles just periodically. Instead, you need call the the `createMissile()` function from a keyboard event handler, which happened to be the `moveRover()` function. (You may have a different name, of course.) Now that we are shooting missiles from this function, it is a bad name, so let's rename the function to something else, say `handleKeyPress()`. (Make sure to change the `onKeyPress` attribute of your main character in the HTML body when you change the name of the function.)

Now, you may want to define another case statement in the `switch(event.keyCode)` clause to execute the function. A slight inconvenience here is that the value `event.keyCode` is always 0, when the pressed key has an ASCII code (which is the case for all the alphabets, and the space key) so, for our purpose, `event.keyCode` is useless other than for detecting the arrow keys.

You can do one of two things: if you are using only one key other than the arrow keys, you can just use the value 0 to shoot the missiles. This will make the game to shoot missiles whenever whatever key is pressed. Or you can use the `event.which` (or `event.charCode`) value, which contains the ASCII code of the key pressed, in case `event.keyCode` is 0. You may want to simply introduce another if statement to handle the situation.

For your information, the ascii code table can be found at: http://ascii.cl/ (and in many other Web sites).

Save the file and reload it on Firefox. Start the game, and shoot a missile. You should see your missile appear where it was specified. You may want to adjust the initial position of the missile at this point.

2.2.5) Defining updateMissiles ()

The definition of this function is similar the definition of `updateObstacles()`, except the modifications already mentioned. Following are the points you need to modify further:

1) You need to move missiles to the opposite direction of obstacles.
2) You need to check the collisions between missiles and **enemies** (and not the main character). This means you need to create another for loop, inside the for loop you probably already have.

3) Once the collision between an enemy and a missile is detected, you can do:
    a. Remove the missile from the document body.
    b. Remove the enemy from the document body.
    c. Increase the score.
4) If you have defined obstacles and enemies, you may also want to define handling of the collision between missiles and obstacles (which means you need yet another for loop!)
5) Once the collision between an obstacle and a missile is detected, you may want to handle it differently the case of the collision between an enemy and a missile. For example, you may want to simply remove the missile from the document body, without removing the obstacle or increasing the score.

2.2.6) Defining clearMissiles ()

The definition of this function is exactly the same as the definition of `clearEnemies()`, except the modifications already mentioned. Make sure this function is called when it is needed. (Where is that?)

2.2.7) Modifying updateGame ()

Insert a function call to execute `updateMissiles()`.

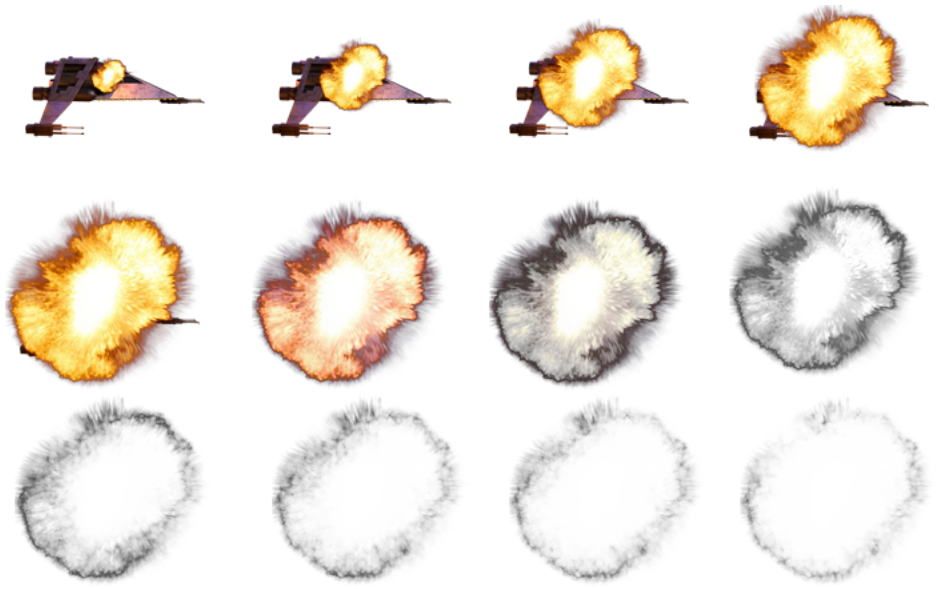
Save the file and reload it on Firefox. Start the game and shoot the missiles. You should see your missiles destroying the enemies.

Bonus Task (5 points): Implementing an animation effect

In this task, you will implement an animation effect when the main character is collided with an obstacle or enemies. The same method can be applied to the enemies, as well, when they are shot down by missiles. There are many ways to do it, so you can do it in your own way, as long as it works. If you came up with a better approach, just let the instructor know. ☺

3.0) Preparation

You will need a set of images that simulates an explosion (or whatever the effect). For example, following are images that were created to show the explosion of the fighter in the example:



Name the files with a numerical postfix, e.g. “fighter_exp1.png”, “fighter_exp2.png”, “fighter_exp3.png”, etc.

If you want to simulate sinking/falling of the main character, you may not need as many images as above. (You will change the position of the images more than changing the image files.)

3.1) Modifying clearGame()

First, initialize a couple of new global variable properties that are needed.

- `gs.roverStatus`

  This property will keep track of the status of the main sprite e.g. 0 means the sprite is normal and 1 means it just collided with an obstacle (or enemy, etc.). Once it becomes 1, it should start counting up. This property should be initialized with 0.

- `gs.roverExpAnimationNum`

  This property holds the last numerical prefix of the graphic image set. (This is just for convenience, and you can live without it…)

3.2) Defining updateRover()

This function, named analogous to the update functions for the other sprites, updates the main character. This function should do the following:

1) Declare a local variable `sp` and initialize it by obtaining the main sprite from the document body.
2) Check if the `gGamestate.roverStatus` is larger than 0 and less than `gGamestate.roverEpxAnimationNum`. This is checking if the main sprite has collided or not.
3) If so (it means the main character has collided with something), do the following steps:
   a. Change the `sp.src` property so that the main character will display the corresponding graphics. For example, you might want to say somthing like:
   ```
   sp.src = "images/fighter_exp" + gGameState.roverStatus + ".png";
   ```

   If you are simulating sinking/falling of the main character, you can change the position of the main sprite similarly.

   b. Increase the `gGamestate.roverStatus` by 1.
4) If not, do nothing.

3.3) Modifying the updateObstacles() / updateEnemies() functions

Now, go to the code segments where the collision between the main sprite and anything is checked. You should see an if statement with a `checkCollision()` function call. You need to modify this if statement in the following way:

1) In the condition, add another condition `gGameState.roverState == 0` using the `&&` operator. Do you know why it is needed?
2) Comment out all the statements in the curly brace clause (but do not delete them because you will need them later) and replace with one statement that assigns 1 to `gGameState.roverState`. This is declaring the collision, and let other functions to handle the situation.

Save the file and reload it on Firefox. Start the game, and try to collide with an obstacle or enemy. The main sprite should explode/sink/fall/etc. Now, all we need to do is put back the logic to handle the death/destruction, etc. of the main character.

3.4) Modifying the updateGame() function

Here, we will put back the logic to handle the death/destruction, etc. of the main character. In the `updateGame()` function, after all the other update functions are called, check if `gGameState.roverState` is larger than or equals to `gGamestate.roverEpxAnimationNum`. This means that the main character has collided and the animation has finished, in other words, time to reset the game. In the if clause, put back the logic that you commented out in the `updateObstacle()` function. Make sure you are calling all the cleaning function (clearObstacles(), clearEnemies(), clearMissiles(), etc.

Save the file and reload it on Firefox. Start the game, and try to collide with an obstacle or enemy. The main sprite should explode/sink/fall/etc, and once the animation is done, the game should handle the death/destruction, etc of the main character properly.

Congratulations! You have finished the LAST LAB ASSIGNMENT of the semester!

Submission

As always, submit your work to MySite. The Instruction of MySite is found at http://its.syr.edu/mysite/.

You will only need to submit the URL to page you created to the discussion board on the ILMS. Create a hyperlink, if you would like. For example, I would submit:

http://kinoue.mysite.syr.edu/IST400-600/Lab11.html

Grading Criteria

The lab assignment must be submitted by the end of Tuesday April 27th. The grading will be based on the following:

- Modification of initializeGame() and clearGame(): 2 points
- Definition of createToken()/createEnemy(): 2 points
- Definition of updateTokens()/updateEnemies(): 2 points
- Definition of clearTokens()/clearEnemies() function: 2 points
- Modification of updateGame() 2 points

Late submission will be accepted within one week from the original due date, with 2 points deduction. 2 points will be deducted for improper submissions (e.g. not using MySite) or for not submitting the peer survey as well.