ASSIGNMENT 1

HTML5 game

nagarro

Nagarro Campus Learning Program

Assignment submission guidelines

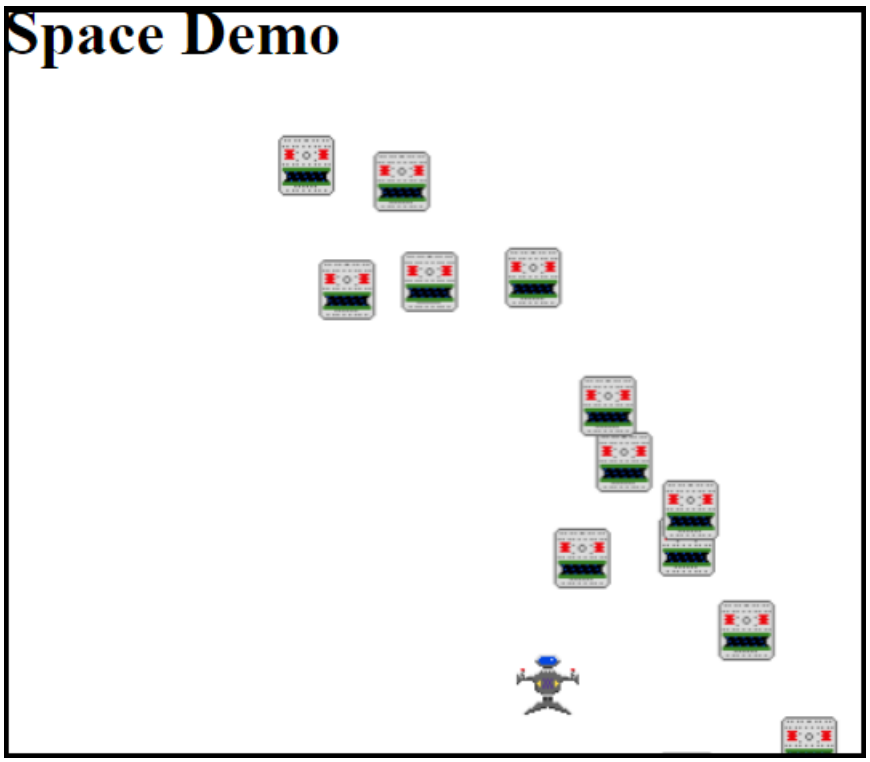
- Follow assignment steps to build the html5 game
- Make sure that following functional pieces are working:
    - Space game is working as per functional details mentioned
    - The Canvas is created
    - Player is created and is firing on press of spacebar
    - Enemies are created and are continuously falling from top
    - Enemies disappear as soon as they are hit with the bullet
    - Additional implementation details mentioned in the end of documents
- Submit following details to nagarro campus team:
    - Original source code
    - Screenshots of working game
    - A small document containing implementation approach and source code organization

Assignment

Space game demo

Functional Details

This is a simple HTML canvas game called Space Demo. It consists of enemies falling from the sky, and a player who must kill them by firing bullets to win the game. Player can move left and right using the arrow keys and can fire bullets using the space bar. Player should make sure that none of the enemies touches the ground, else he will lose. Once you have gone through the assignment and learnt how to make a game with HTML Canvas, we have added few interesting challenges for you !!!



Step by step guide to build the html5 game:

Initial Setup

We have already created a project with complete folder structure to start with. It also contains few JavaScript libraries that we thought were difficult to download and for the rest we have provided download links in the tutorial. We have also provided the required resources such as images of enemies, player and sounds required for the tutorial in the respective folders. Steps to use the project are as follows:

- Right click on the below zip file and save it on your desktop.
- Unzip the file.
- Open the folder and look for Index.html

- Right click on Index.html and open with chrome (your browser). It should show “Hello world”. We are all set to start!!!

space_demo.zip

Step 1

Creating the canvas

The HTML <canvas> element is used to draw graphics, on the fly, via JavaScript. The <canvas> element is only a container for graphics. You must use JavaScript to draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

Reference link: http://html5gamedevelopment.com/html5-simple-canvas-tutorial/

In order to draw things, we'll need to create a canvas. We'll be using jQuery. You can download jQuery from https://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js and save it to scripts folder.

```
var CANVAS_WIDTH = 480;

var CANVAS_HEIGHT = 320;



var canvasElement = $("<canvas width='" + CANVAS_WIDTH +

                      "' height='" + CANVAS_HEIGHT +
"'></canvas>");

var canvas = canvasElement.get(0).getContext("2d");

canvasElement.appendTo('body');
```

Step 2

Using setInterval() for Game loop

The setInterval() method calls a function or evaluates an expression at specified intervals (in milliseconds). The setInterval() method will continue calling the function until clearInterval() is called, or the window is closed.The ID value returned by setInterval() is used as the parameter for the clearInterval() method.

Reference link: http://www.w3schools.com/jsref/met_win_setinterval.asp

Why set interval?

In order to simulate the appearance of smooth and continuous gameplay, we want to update the game and redraw the screen just faster than the human mind and eye can perceive.

```
var FPS = 30;

setInterval(function() {

  update();

  draw();

}, 1000/FPS);
```

For now we can leave the update and draw methods blank. The important thing to know is that setInterval() takes care of calling them periodically.

```
function update() { ... }

function draw() { ... }
```

Hello world

Now that we have a game loop going, let's update our draw method to actually draw some text on the screen.

```
function draw() {

  canvas.fillStyle = "#000"; // Set color to black

  canvas.fillText("Sup Bro!", 50, 50);

}
```

Pro Tip: Be sure to run your app after making changes. If something breaks it's a lot easier to track down when there's only a few lines of changes to look at.

That's pretty cool for stationary text, but because we have a game loop already set up, we should be able to make it move quite easily

```
var textX = 50;

var textY = 50;



function update() {

  textX += 1;

  textY += 1;

}



function draw() {

  canvas.fillStyle = "#000";

  canvas.fillText("Sup Bro!", textX, textY);

}
```

Now give that a whirl. If you're following along, it should be moving, but also leaving the previous times it was drawn on the screen. Take a moment to guess why that may be the case. This is because we are not clearing the screen. So let's add some screen clearing code to the draw method.

```
function draw() {

  canvas.clearRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);

  canvas.fillStyle = "#000";

  canvas.fillText("Sup Bro!", textX, textY);

}
```

Now that you've got some text moving around on the screen, you're halfway to having a real game. Just tighten up the controls, improve the gameplay, touch up the graphics.... Ok maybe 1/7th of the way to having a real game, but the good news is that there's much more to the tutorial.

Step 3

Creating the player

Create an object to hold the player data and be responsible for things like drawing. Here we create a player object using a simple object literal to hold all the info.

```
var player = {

  color: "#00A",

  x: 220,

  y: 270,

  width: 32,

  height: 32,

  draw: function() {

    canvas.fillStyle = this.color;

    canvas.fillRect(this.x, this.y, this.width, this.height);

  }

};
```

We're using a simple colored rectangle to represent the player for now. When we draw the game, we'll clear the canvas and draw the player.

```
function draw() {

  canvas.clearRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);

  player.draw();

}
```

Step 4

Keyboard controls

Using jQuery Hotkeys

You can download jquery.hotkeys.js from https://plugins.jquery.com/hotkeys/ and save it to scripts folder.

The jQuery Hotkeys plugin makes key handling across browsers much much easier. Rather than crying over indecipherable cross-browser keyCode and charCode issues, we can bind events like so:

```
$(document).bind("keydown", "left", function() { ... });
```

Not having to worry about the details of which keys have which codes is a big win. We just want to be able to say things like "when the player presse the up button, do something." jQuery Hotkeys allows that nicely.

Reference link: https://plugins.jquery.com/hotkeys/

Player movement

The way JavaScript handles keyboard events is completely event driven. That means that there is no built in query for checking whether a key is down, so we'll have to use our own.

You may be asking, "Why not just use an event-driven way of handling keys?" Well, it's because the keyboard repeat rate varies across systems and is not bound to the timing of the game loop, so gameplay could vary greatly from system to system. To create a consistent experience, it is important to have the keyboard event detection tightly integrated with the game loop.

We've included a 16-line JS wrapper that will make event querying available. It's called key_status.js and you can query the status of a key at any time by checking keydown.left, etc.

You can download key_status.js from https://sites.google.com/a/bay.k12.fl.us/nbhca-technology/space-shooter-2 and save it to scripts folder.

Now that we can query whether keys are down, we can use this simple update method to move the player around.

```
function update() {

  if (keydown.left) {

    player.x -= 2;

  }



  if (keydown.right) {

    player.x += 2;

  }

}
```

Go ahead and give it a whirl.

You might notice that the player is able to be moved off of the screen. Let's clamp the player's position to keep them within the bounds. Additionally, the player seems kind of slow, so let's bump up the speed, too.

```
function update() {

  if (keydown.left) {

    player.x -= 5;

  }



  if (keydown.right) {

    player.x += 5;

  }



  player.x = player.x.clamp(0, CANVAS_WIDTH - player.width);

}
```

Adding more inputs will be just as easy, so let's add some sort of projectiles.

```
function update() {

  if (keydown.space) {

    player.shoot();

  }



  if (keydown.left) {

    player.x -= 5;

  }
```

```
  if (keydown.right) {
    player.x += 5;
  }

  player.x = player.x.clamp(0, CANVAS_WIDTH - player.width);
}

player.shoot = function() {
  console.log("Pew pew");
  // :) Well at least adding the key binding was easy...
};
```

Step 5

Adding more game objects

Projectiles

Let's now add the projectiles for real. First, we need a collection to store them all in:

```
var playerBullets = [];
```

Next, we need a constructor to create bullet instances.

```
function Bullet(I) {

  I.active = true;



  I.xVelocity = 0;

  I.yVelocity = -I.speed;

  I.width = 3;

  I.height = 3;

  I.color = "#000";
```

```
  I.inBounds = function() {

    return I.x >= 0 && I.x <= CANVAS_WIDTH &&

      I.y >= 0 && I.y <= CANVAS_HEIGHT;

  };



  I.draw = function() {

    canvas.fillStyle = this.color;

    canvas.fillRect(this.x, this.y, this.width, this.height);

  };



  I.update = function() {
```

```
    I.x += I.xVelocity;

    I.y += I.yVelocity;



    I.active = I.active && I.inBounds();

  };



  return I;

}
```

When the player shoots, we should create a bullet instance and add it to the collection of bullets.

```
player.shoot = function() {

  var bulletPosition = this.midpoint();



  playerBullets.push(Bullet({

    speed: 5,

    x: bulletPosition.x,

    y: bulletPosition.y

  }));
```

```
};



player.midpoint = function() {

  return {

    x: this.x + this.width/2,

    y: this.y + this.height/2

  };

};
```

We now need to add the updating of the bullets to the update step function. To prevent the collection of bullets from filling up indefinitely, we filter the list of bullets to only include the active bullets. This also allows us to remove bullets that have collided with an enemy.

```
function update() {

  ...

  playerBullets.forEach(function(bullet) {

    bullet.update();

  });



  playerBullets = playerBullets.filter(function(bullet) {

    return bullet.active;

  });

}
```

The final step is to draw the bullets:

```
function draw() {

  ...

  playerBullets.forEach(function(bullet) {

    bullet.draw();

  });

}
```

Enemies

Now it's time to add enemies in much the same way as we added the bullets.

```
  enemies = [];



function Enemy(I) {

  I = I || {};



  I.active = true;
```

```
I.age = Math.floor(Math.random() * 128);



I.color = "#A2B";



I.x = CANVAS_WIDTH / 4 + Math.random() * CANVAS_WIDTH / 2;

I.y = 0;

I.xVelocity = 0

I.yVelocity = 2;



I.width = 32;

I.height = 32;
```

```
I.inBounds = function() {

  return I.x >= 0 && I.x <= CANVAS_WIDTH &&

    I.y >= 0 && I.y <= CANVAS_HEIGHT;

};



I.draw = function() {
```

```
    canvas.fillStyle = this.color;

    canvas.fillRect(this.x, this.y, this.width, this.height);

  };



  I.update = function() {

    I.x += I.xVelocity;

    I.y += I.yVelocity;



    I.xVelocity = 3 * Math.sin(I.age * Math.PI / 64);



    I.age++;
```

```
    I.active = I.active && I.inBounds();

  };



  return I;

};
```

```
function update() {

  ...



  enemies.forEach(function(enemy) {

    enemy.update();

  });



  enemies = enemies.filter(function(enemy) {

    return enemy.active;

  });
```

```
  if(Math.random() < 0.1) {
    enemies.push(Enemy());
  }
};

function draw() {
  ...

  enemies.forEach(function(enemy) {
    enemy.draw();
  });
}
```

Step 6

Loading and drawing images

It's cool watching all those boxes flying around, but having images for them would be even cooler. Loading and drawing images on canvas is usually a tearful experience. To prevent that pain and misery, we can use a simple utility class.

sprite.js and util.js are already present in the scripts folder.

Reference Link : https://spritejs.readthedocs.io/en/latest/introduction.html

```
player.sprite = Sprite("player");



player.draw = function() {

  this.sprite.draw(canvas, this.x, this.y);

};



function Enemy(I) {

  ...
```

```
  I.sprite = Sprite("enemy");



  I.draw = function() {

    this.sprite.draw(canvas, this.x, this.y);

  };



  ...

}
```

Step 7

Collision detection

We've got all these dealies flying around on the screen, but they're not interacting with each other. In order to let everything know when to blow up, we'll need to add some sort of collision detection.

Let's use a simple rectangular collision detection algorithm:

```
function collides(a, b) {

  return a.x < b.x + b.width &&

         a.x + a.width > b.x &&

         a.y < b.y + b.height &&

         a.y + a.height > b.y;

}
```

There are a couple collisions we want to check:

1. Player Bullets => Enemy Ships
2. Player => Enemy Ships

Let's make a method to handle the collisions which we can call from the update method.

```
function handleCollisions() {

  playerBullets.forEach(function(bullet) {

    enemies.forEach(function(enemy) {

      if (collides(bullet, enemy)) {

        enemy.explode();

        bullet.active = false;

      }

    });

  });



  enemies.forEach(function(enemy) {

    if (collides(enemy, player)) {

      enemy.explode();

      player.explode();
```

Now we need to add the explode methods to the player and the enemies. This will flag them for removal and add an explosion.

```
function Enemy(I) {

  ...



  I.explode = function() {

    this.active = false;

    // Extra Credit: Add an explosion graphic

  };
```

```
  return I;

};



player.explode = function() {

  this.active = false;

  // Extra Credit: Add an explosion graphic and then end the game

};
```

Step 8

Sound

To round out the experience, we're going to add some sweet sound effects. Sounds, like images, can be somewhat of a pain to use in HTML5, but thanks to sound.js, sound can be made super-simple.

Sound.js is already present in the scripts folder.

```
player.shoot = function() {

  Sound.play("shoot");

  ...

}



function Enemy(I) {

  ...
```

```
  I.explode = function() {

    Sound.play("explode");

    ...

  }

}
```

Adding sounds is currently the quickest way to crash your application. It's not uncommon for sounds to cut-out or take down the entire browser tab, so get your tissues ready.

Additional important implementations

Make sure following additional implementation is working along with above basic implementation:

Note
Please note these are most important implementation and carry height weightages

Scoring

Create Score on each successful enemy kill, for e.g. 10 points for every successful hit should be displayed on the top right corner of the screen.

Game over after 3 enemies

Make the player loose if the enemy hits the player or more than 3 enemies cross the player.

New enemy type

Add another enemy in the game of your own choice, with a weightage of 20 points and manage score considering both the enemies.

Levels

Increase the level, by increasing the speed of falling enemies.