Tutorial – Basic Linear Math and Collision Detection

In this week's tutorial we'll take the code you made last week and update it to use a velocity vector and a speed scalar.

We'll also add an asteroid to the scene and allow the player to fire a bullet by pressing the space bar. The AABB collision detection algorithm mentioned in the lecture will be used to detect when the bullet hits the asteroid.

Velocity Vector and Speed:

As we start adding more and more functionality to our game, things are going to start getting a little messy. So let's start by cleaning up our code a little.

The first thing we'll do is create a player object with a few properties.

Taking your code from last week, modify it follows:

1. Remove the player, x, y, and angle variable definitions and replace them with the following object definition:

```
~~var player = document.createElement("img");~~
~~player.src = "ship.png";~~

~~var x = 10;~~
~~var y = 10;~~
~~var angle = 0;~~

      // this creates the player object and assigns it some properties
var player = {
      image: document.createElement("img"),
      x: SCREEN_WIDTH/2,
      y: SCREEN_HEIGHT/2,
      width: 93,
      height: 80,
      directionX: 0,
      directionY: 0,
      angularDirection: 0,
      rotation: 0
};
      // set the image for the player to use
player.image.src = "ship.png";
```

What we've done here is created a player 'object'. This object is a single variable that can have any number of properties. Objects can even have their own functions (we'll get to that during assignment 2).

This is a little similar to what we had before, except now instead of an x and y variable, we use the player.x and player.y variable. This makes our code much easier to understand.

Note that the direction refers to whether or not the player is moving forwards or backwards. This direction vector is relative to the player's local coordinate space (if y is +1, then that means forward, regardless of how the ship is rotated).

2. While we're updating the player, let's add some more constant variables.

   Add these values just above the player definition:

```
var SCREEN_WIDTH = canvas.width;
var SCREEN_HEIGHT = canvas.height;

var ASTEROID_SPEED = 0.8;
var PLAYER_SPEED = 1;
var PLAYER_TURN_SPEED = 0.04;
var BULLET_SPEED = 1.5;
```

3. onKeyDown and onKeyUp will need to change. Instead of calculating how much to modify the x and y position, we'll just set the Y direction to positive or negative. Then, during the run() function, we'll calculate the current velocity based on the direction and rotation, and apply that to the player's position.

```
function onKeyDown(event)
{
	if(event.keyCode == KEY_UP)
	{
		player.directionY = 1;
	}
	if(event.keyCode == KEY_DOWN)
	{
		player.directionY = -1;
	}
	if(event.keyCode == KEY_LEFT)
	{
		player.angularDirection = -1;
	}
	if(event.keyCode == KEY_RIGHT)
	{
		player.angularDirection = 1;
	}
}

function onKeyUp(event)
{
	if(event.keyCode == KEY_UP)
	{
		player.directionY = 0;
	}
	if(event.keyCode == KEY_DOWN)
	{
		player.directionY = 0;
	}
	if(event.keyCode == KEY_LEFT)
	{
		player.angularDirection = 0;
	}
	if(event.keyCode == KEY_RIGHT)
	{
		player.angularDirection = 0;
	}}
```

4. Finally, update the run function so that we update the player's position based on the current direction and speed (i.e., the velocity) of the player.

```
function run()
{
	context.fillStyle = "#ccc";
	context.fillRect(0, 0, canvas.width, canvas.height);

		// calculate sin and cos for the player's current rotation
	var s = Math.sin(player.rotation);
	var c = Math.cos(player.rotation);

	// figure out what the player's velocity will be based on the current rotation
	// (so that if the ship is moving forward, then it will move forward according to its
	// rotation. Without this, the ship would just move up the screen when we pressed 'up',
	// regardless of which way it was rotated)
	// for an explanation of this formula, see http://en.wikipedia.org/wiki/Rotation_matrix
	var xDir = (player.directionX * c) - (player.directionY * s);
	var yDir = (player.directionX * s) + (player.directionY * c);
	var xVel = xDir * PLAYER_SPEED;
	var yVel = yDir * PLAYER_SPEED;

	player.x += xVel;
	player.y += yVel;

	player.rotation += player.angularDirection * PLAYER_TURN_SPEED;

	context.save();
		context.translate(player.x, player.y);
		context.rotate(player.rotation);
		context.drawImage(
			player.image, -player.width/2, -player.height/2);
	context.restore();
}
```

If you run the game now you'll see that it pretty much does exactly what it did last week. So what did we do and why did we bother doing it?

Well, beside from cleaning things up a bit, we've change the code so that now we use a direction vector and a speed variable (which we use to calculate a velocity vector during the run() function).

When we hold down the up key, the y direction is set to 1. The direction vector should always be normalized but since we're dealing only with the y direction (we can't move from left to right) that will always be the case.

During the run() function we take the current direction and use the player's current rotation and speed to calculate the x and y velocity.

We then add the velocity to the player's current position to get the new x and y position of the player.

This method of updating is much more flexible than what we did last week, since we can now change the player's speed (dynamically if we wanted to), or map different keys to modify the x and y direction of the player.

This is also the base code we'll use in assignment 2, where we'll add acceleration and deceleration.

Adding an Asteroid:

We'll create a new asteroid using the same approach we used to create the player.

1. Create the asteroid object and give it some properties

```
	// this creates the asteroid object and assigns it some
	// properties
var asteroid = {
	image: document.createElement("img"),
	x: 100,
	y: 100,
	width: 69,
	height: 75,
	directionX: 0,
	directionY: 0
};
	// set the image for the asteroid to use
asteroid.image.src = "rock_large.png";
```

2. Next, let's set the asteroid's x and y direction values. Remember, the direction must be normalized because we'll multiply it by the asteroid's speed in the run() function to calculate the asteroid's velocity.

```
asteroid.directionX = 10;
asteroid.directionY = 8;
	// 'normalize' the velocity (the hypotenuse of the triangle
	// formed by x,y will equal 1)
var magnitude = Math.sqrt( (asteroid.directionX * asteroid.directionX)
		+ (asteroid.directionY * asteroid.directionY) );
if(magnitude != 0)
{
	asteroid.directionX /= magnitude;
	asteroid.directionY /= magnitude;
}
```

3. Finally, in the run() function add some code to update and draw the asteroid.

   It's a bit simpler than the player's update code because we're not worrying about rotating the asteroid.

   Add the following code to the **end** of the run() function.

```
// update and draw the asteroid (we don't need to
// worry about rotation here)
var velocityX = asteroid.directionX * ASTEROID_SPEED;
var velocityY = asteroid.directionY * ASTEROID_SPEED;
asteroid.x += velocityX;
asteroid.y += velocityY;
context.drawImage(asteroid.image,
      asteroid.x - asteroid.width/2,
      asteroid.y - asteroid.height/2);
```

You should now have a player ship that you can move around the screen, and an asteroid that slowly drifts off the screen.

Next we'll add a bullet object so we can shoot that pesky asteroid.

Firing a Bullet:

This is going to be very similar to what we just did for the asteroid, except we'll wait until the player has fired before setting the bullet's direction.

For the moment we'll only let the player fire one bullet at a time (so only one bullet will ever be on the screen at any time). Next week we'll expand on this code to allow multiple asteroids and multiple bullets.

1. After the code that creates the asteroid (before the onKeyDown() function), add the following code to create the bullet object:

```
        // this creates the bullet object and assigns it some properties
var bullet = {
        image: document.createElement("img"),
        x: player.x,
        y: player.y,
        width: 5,
        height: 5,
        velocityX: 0,
        velocityY: 0,
        isDead: true
};
        // set the image for the asteroid to use
bullet.image.src = "bullet.png";
```

2. Now add the following function to allow the player to shoot a bullet:

```
function playerShoot()
{
	// we can only have 1 bullet at a time
	if( bullet.isDead == false )
		return;

	// start off with a velocity that shoots the bullet straight up
	var velX = 0;
	var velY = 1;

	// now rotate this vector acording to the ship's current rotation
	var s = Math.sin(player.rotation);
	var c = Math.cos(player.rotation);

	// for an explanation of this formula,
	// see http://en.wikipedia.org/wiki/Rotation_matrix
	var xVel = (velX * c) - (velY * s);
	var yVel = (velX * s) + (velY * c);

	// dont bother storing a direction and calculating the
	// velocity every frame, because it won't change.
	// Just store the pre-calculated velocity
	bullet.velocityX = xVel * BULLET_SPEED;
	bullet.velocityY = yVel * BULLET_SPEED;

	// don't forget to set the bullet's position
	bullet.x = player.x;
	bullet.y = player.y;

	// make the bullet alive
	bullet.isDead = false;
}
```

3. We also need to update the onKeyUp() function so that when the space bar is released the ship will fire a bullet.

   We could put this code in the onKeyDown() function, but by using the key up event we ensure that only one bullet will fire regardless of how long the player presses the fire key. (That's not a problem right now because we are checking if the bullet has already been fired, but next week when we add multiple bullets it might cause us problems depending on the functionality we want).

Add the following code to the **end** of the onKeyUp() function:

```
function onKeyUp(event)
{
	. . .

	if(event.keyCode == KEY_SPACE)
	{
		playerShoot();
	}
}
```

4. And finally we can update the run() function to update and draw the bullet.

   We'll place this code at the top of the run() function (just after we clear the screen) so that the bullet is drawn under the player (anything you draw later is drawn over the top of whatever you drew earlier).

```
function run()
{
	context.fillStyle = "#ccc";
	context.fillRect(0, 0, canvas.width, canvas.height);

	// update and draw the bullet
	// we want to do this first so that the plane is drawn on
	// top of the bullet
	if(bullet.isDead == false)
	{
		bullet.x += bullet.velocityX;
		bullet.y += bullet.velocityY;
		context.drawImage(bullet.image,
			bullet.x - bullet.width/2,
			bullet.y - bullet.height/2);
	}

	. . .
}
```

Adding Collision Checks:

The final thing we need to do is get the bullet colliding with the ship. We will also need to kill the bullet if it goes off the screen so that the player is able to fire again.

To add the collision checking, we're going to use the formula described in the lecture. You should make sure you understand the AABB method to test for the intersection of two rectangles.

1. Add the following function just before the run() function:

```
// tests if two rectangles are intersecting.
// Pass in the x,y coordinates, width and height of each rectangle.
// Returns 'true' if the rectangles are intersecting
function intersects(x1, y1, w1, h1, x2, y2, w2, h2)
{
	if(y2 + h2 < y1 ||
		x2 + w2 < x1 ||
		x2 > x1 + w1 ||
		y2 > y1 + h1)
	{
		return false;
	}
	return true;
}
```

2. Add the isDead variable to the definition of the asteroid object:

```
var asteroid = {
	image: document.createElement("img"),
	x: 100,
	y: 100,
	width: 69,
	height: 75,
	directionX: 0,
	directionY: 0,
	isDead: false
};
```

3. Modify the bullet updating logic in the run() function as follows:

```
function run()
{
	context.fillStyle = "#ccc";
	context.fillRect(0, 0, canvas.width, canvas.height);

	// update and draw the bullet
	// we want to do this first so that the plane is drawn on
	// top of the bullet
	if(bullet.isDead == false)
	{
		bullet.x += bullet.velocityX;
		bullet.y += bullet.velocityY;
		context.drawImage(bullet.image,
			bullet.x - bullet.width/2,
			bullet.y - bullet.height/2);

		if(asteroid.isDead == false)
		{
			var hit = intersects(
					bullet.x, bullet.y,
					bullet.width, bullet.height,
					asteroid.x, asteroid.y,
					asteroid.width, asteroid.height);
			if(hit == true)
			{
				bullet.isDead = true;
				asteroid.isDead = true;
			}
		}

		if(bullet.x < 0 || bullet.x > SCREEN_WIDTH ||
			bullet.y < 0 || bullet.y > SCREEN_HEIGHT)
		{
			bullet.isDead = true;
		}
	}

	. . .
}
```

Congratulations! If everything went smoothly you should now have a ship and an asteroid in your game, and you should be able to fire a bullet and shoot the asteroid.

Exercise:

Can you modify the game so that when the asteroid hits the player, the player will die?

See if you and also add some code so that when the asteroid moves off one side of the screen it will warp around and enter from the other side of the screen.