BattleBeans | battlebeansgame.com

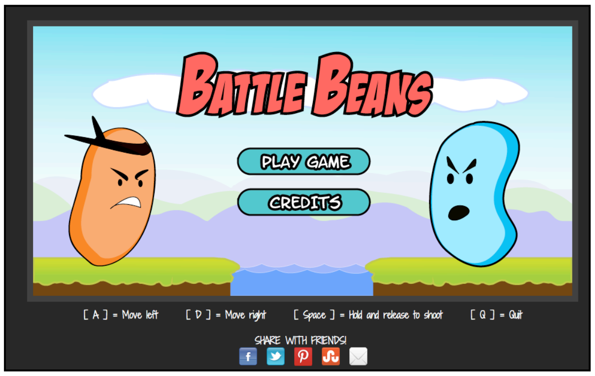
Dylan Barth, An Yu, Julie Huynh
Cogs 121, Spring 2012

I. Project Overview

The goal of our project was to create an HTML5 canvas game using JavaScript. Originally, we envisioned a game similar to Pocket Tanks, where two characters would participate in a turn based game that involves firing projectiles toward an enemy until one or the other is destroyed. In the initial stages of our planning, we thought that trying to build a turn-based game would allow us to slow things down and control every event closely. However, as we started building the game, we realized that even though we probably could build a turn-based game, we could leverage JavaScript's event handling power to create a similar style of game that occurred in real-time. This idea of a faster paced game with more interactivity for the user really attracted us, and that turned out to be the core idea that made BattleBeans into what it is at this point.

II. Description of Interface & Application

BattleBeans is an interactive 1-player game written in JavaScript, HTML5, and CSS. The game involves two fictitious bean characters (one controlled by the user, one computer player) that battle with one another in real time by lobbing bean bullets across the screen. The first bean to score three hits on the opponent wins the game.



Landing Page

The game's landing page consists of a splash screen, instructions for the control of the player's bean, and social media sharing buttons. There are two options on the splash screen menu: 'Play Game' and 'Credits'.

The Game

When the game starts, the game loop is initialized. The enemy bean starts moving and firing immediately. The player can move the pinto bean character left and right (using keys A and D) across the 'ground' to dodge the incoming bean bullets. Both beans are only able to move a certain distance to the left or the right, hindered by the edge of the canvas and the edge of the cliff.

To fire a bullet, the player holds the spacebar key down to control the distance the bean bullet lands. A green power bar appears on the left to provide a relative indication of how far the bullet will go. When the spacebar is released, a bean bullet comes out of the bean and is launched in an arc.

If a bean is hit, its expression changes for a few seconds, and one of its lives (depicted by smaller versions of the bean near the top) disappears. When either runs out of lives, the game loop is stopped and the game is over. If the player is the winner, congratulatory music plays. If the player loses, a heart rate monitor flat line sound comes on. The player can press 'R' to restart the game at this point. During the course of the game, the player can press 'Q' to end the game early and bring up the splash screen at any point.

Credits
The credits page is a static page that shows comic-styled versions of us, the game creators.

III. Discussion of Design Process & Current State

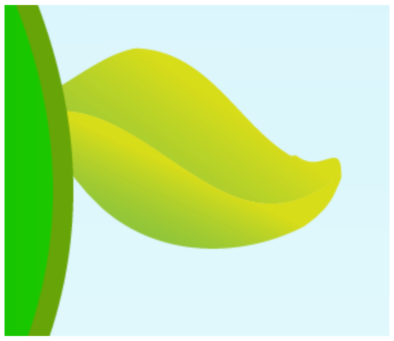
We started with just a general idea of what we wanted to create and a beginner's understanding of JavaScript. We broke down the features of the game into the core components and the peripheral components. We set weekly goals to complete specific features based on what we thought was a realistic timeline.

As we implemented features, some things worked better than we thought they would, and others didn't. In the end, we had to leave out some of our planned features (e.g. adding elevation and multiplayer modes). The game changed as we realized what JavaScript could do well (e.g. handling events).

***See 'Expectations vs. Reality Timeline' in Appendix**

Once we got a good solid game working, we brainstormed as many cool improvements as we could. We prioritized the list by what we thought we could accomplish and what would improve the game the most. That meant that the complete control of the shooting feature came before other features, like sounds and improved graphics.

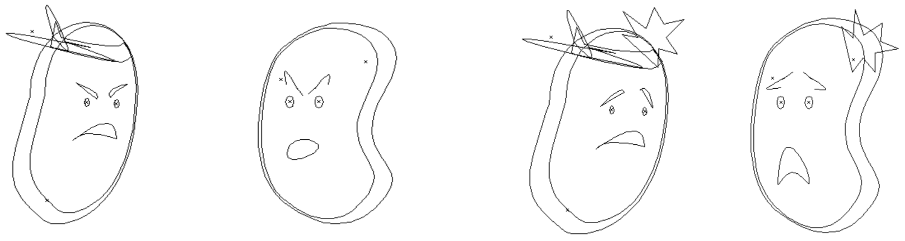
Lots of our inspiration for the game's graphic style came from looking at existing games (especially iPhone games) and noting the styles we liked. We decided we wanted a cute vector theme similar to Angry Birds and Little Acorns, so we went through tutorials to learn how to use the Pen and Warps tools in Adobe Illustrator to create landscapes and characters.

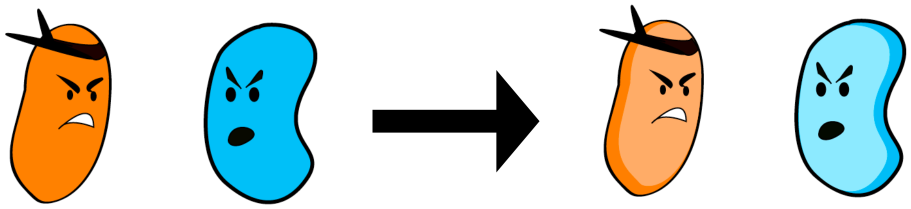


Our first landscape had clouds with beanstalks, but we realized that the bean characters only moved horizontally. Since they only moved horizontally, the beans never crossed one another on the landscape. So we created a second landscape that implemented a division between the two bean characters. We created two cliffs that were divided by a body of water. That way, the beans never crossing made sense with the landscape. Another tool we had to learn was the gradient tool. This tool gave the 3-D effect to our landscape;

otherwise our vector images would have looked flat. After reading tutorials on the gradient tool, we were able to split the beanstalk leaves in half and create a separate gradient for each half. Using this technique, we were able to easily create the depth in the sky and cliffs.

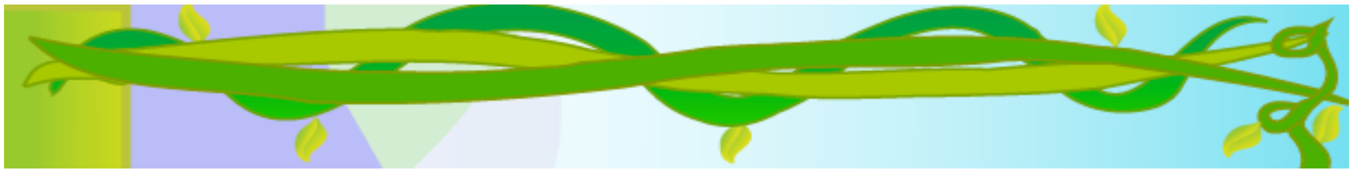
The bean characters were created by manipulating Bézier curves with the Pen Tool and then applying fill color and strokes to make them look cartoony.





We looked at cartoon characters to see what made them look 3-D, and found that adding a lighter shade of the same color made a huge difference in bringing more depth to the illustrations. Made sense, as it's basically creating a shadow effect.

Since we had a working power bar, we had to create new graphics for our game. We needed a way to show the power levels on the landscape, either vertically or horizontally. We decided to put the levels vertically on the left and right sides of the screen. We had the idea of creating beanstalks underneath the power levels, and expanding them with peapods above them. The peapods would lay inward of the beanstalks so that they could hold the "health" of the characters. Each pea in the peapod were the health "hearts". We struggled with creating the beanstalks because this required precise detail of the Pen Tool in Illustrator. We started using the Warp Tool, which mainly helped with the previous two landscapes. After we were able to create the shape of the beanstalks, we had to choose the right colors to fill them. This was also difficult because the power levels had to show up without being overpowered by the beanstalks. As for the peapods, we ended up removing the peapods because it didn't look as we imagined. So, we enlarged the bean characters and used them as health points. The actual stalks were difficult to interweave, so we ended up overlaying them rather than weaving them.



We decided to display the game controls below the canvas (instead of as a separate menu item) because we felt that people generally dive into games without bothering to navigate to the instructions. Having the controls visible with the game prevents people from trying to figure it out on their own and getting confused.

IV. Code Documentation

Overview

The game runs on about 500 lines of JavaScript. It's fairly easy to understand once you know what each chunk of code does, and we tried to organize it in an intuitive way. Basically, there are two main chunks of code: stuff outside initialize() and stuff inside initialize(). The outside code controls everything that happens outside of the game play, including the menu, handling the user pressing R to restart, and establishing global variables like the sound effects.

Once 'Play Game' is clicked, initialize() is called and the game loop starts. In a nutshell, the game runs by calling two functions over and over - update() and draw(). Update() checks for player input, changes the positions of the characters, calculates bullet trajectories, keeps track of health points, and checks for impact between bullets and beans. Draw() simply redraws the images on top of the previous images.

When the health points of one bean gets to 0, update() calls endGame() which exits the loop, shows the end of the game text, plays the appropriate sound effect, and prompts the user to play again by pressing R.

Now that we have established the big parts of the code structure, let's delve into the specifics.

Our Game Objects

There are six object constructors that create the landscape, beans, power bars, bullets, and health points. Each object has different properties that determine its position and state. Take the bean constructor for example:

```
function Bean(x) {
        this.x = x;
        this.y = 350;
        this.height = 75;
        this.width = 50;
        this.health = 3;
}
```

The x position and y position determine where the bean is on the canvas. Note that the bean constructor must be passed an x position, but the y position is hard coded in. This ensures that the 'ground' is at y = 350 on the canvas, and allows different beans to be drawn in different places on the canvas on that axis. The height and weight give it an area that can be cross-referenced with the bullet's position to check for impact. The health is simply the number of lives a bean has. On impact, one health point is subtracted.

Drawing Images & Image Control

Notably absent in our object constructors is an image path. Originally, we attempted to include this as part of the constructor, but we kept being tripped up by two main issues. To explain, let's explore an example:

```
/*++++++++Beans++++++++*/
    var bean1Ready = false;
    var bean1Image = new Image();
    bean1Image.onload = function() {
        bean1Ready = true;
    };
    bean1Image.src = "images/bean1.png";
```

First, the larger the image, the slower it loads in the browser. That meant that we had to use image.onload to prevent the browser from attempting to draw the image before it loaded. Putting this function inside the constructor created a problem with scope, because we were then unable to access the state of the image (E.g. we couldn't change the state of bean1.imageReady in order to 'kill' the bean when its health reached 0). Another problem was that loading all of our images at once normally resulted in the beans and bullets being drawn behind the canvas (which made for quite a boring game). We tried a lot of different workaround but eventually just created the 'Image Control Center', which loaded the images separately and created global flags that we could use to draw and undraw the images at our leisure. It wasn't the most elegant solution, but it was very convenient and it allowed us to focus our attention on building other parts of the game.

The Menu

We have three different views that the user can access within the canvas: the 'splash screen' (which contains the menu), the credits page (just a static page with our pictures on it), and the game screen itself. When the page loads, all three of these screens are actually loaded, but credits and the game are hidden using CSS. When a user clicks one of the two menu options, the beginning of our game.js file toggles the appropriate views.

For example: when a user clicks 'Credits', a line in our JavaScript code sets the splash screen to 'hidden' and the credits page to 'block'.

```
var credits_button = document.getElementById('credits-button');

credits_button.onclick = function() {
    splash.style.display = 'none';
    credits_page.style.display = 'block'
};
```

The Game Loop

Once initialize() is called, the game begins by a call to newGame() which starts the appropriate music and then sets the gameLoop, game(), to run as fast as possible. As mentioned previously, game just calls update() and draw() over and over again until endGame() is called.

```
        var game = function() {
                update();
                draw();
            }

        function newGame() {
            gameLoop = setInterval(game, 1);
            themeMusic.currentTime = 0;
            themeMusic.loop = true;
            themeMusic.play();
            loss.pause();
            win.pause();
        }
        newGame();
    }
```

Update

Before delving into the specifics, a quick note about update(). Basically, update is changing the properties of everything. If the player is pressing down a key, update changes the x position of the bean or tells the power bar to start powering up or tells the bullet to shoot. It calculates whether or not the enemy bean is moving right or left or shooting or powering up. It checks to see if a bullet has hit a target and if it has, whether or not the game should end. All of this is accomplished by boolean variables.

For example:

```
if (bullet.fire) {
          powerBar = false;
          bar.height = 0;
          bar.length = 0;
          bulletReady = true;
}
```

In this example, once the player has released the spacebar, bullet.fire is set equal to true. The powerBar is set to false and the positions are reset. Then, bulletReady is set to true. Now let's see what happens when these changes are applied within draw().

Draw

So after update() makes its checks and changes, draw() makes those changes appear on the screen. Following the example above, when a player has released the spacebar.

```
if (powerBar) {
                context.beginPath();
                context.rect(bar.x, bar.y, bar.width, bar.height);
                context.fillStyle = bar.color;
                context.fill();
                context.lineWidth = 1;
                context.strokeStyle = "black";
                context.stroke();
            }
```

Draw knows not to draw the power bar, because powerBar is false so it will not pass the conditional filter. Next:

```
if (bulletReady) {
      context.drawImage(bulletImage, bullet.x, bullet.y);
}
```

Now that bulletReady = true, the bullet will appear on the canvas. The next iteration, update() will know that bullet.fire = true, and will start to plug the bullet's position into the sine function (see 'Launching Bean Bullets' below).

Player Movement

The play controls their bean by pressing A for left, D for right, and space for fire. Outside of update, we set event listeners to always be on the lookout for these keys being pressed. If they were, we added their keyCode to an object we called keyCode (really creative, we know).

```
addEventListener("keydown", function(e) {
    pushed[e.keyCode] = true;
}, false);

addEventListener("keyup", function(e) {
    delete pushed[e.keyCode];
}, false);
```

This way, instead of executing commands immediately, we could control the flow of the gameplay. Inside of update, we check inside the object keyCode for different keys that had been pressed:

```
if (65 in pushed && bean1.x > 35) { // left
    bean1.x -= 3;
}
if (68 in pushed && bean1.x < 300) { // right
     bean1.x += 3;
}
```

If the respective key was found within the object, the bean's position was updated. The same logic worked in reverse (e.g. keyup instead of keydown) for handling shooting.

Enemy Bean Movement

The enemy bean's behavior is completely randomized. The bean's movement is controlled by a boolean that is established at the very beginning of the game:

```
var initial_bean2 = (Math.floor(Math.random() * 2));
var counter = 0; //used later for update loop
if (initial_bean2 == 1) {
    var bean2_right = true;
}
if (initial_bean2 == 0) {
    var bean2_right = false;
}
```

In the code above, bean2_right is given a 50-50 chance of being true or false. If true, the enemy bean starts by moving right, and vice versa. Once the initial direction is established, the bean continues in that direction until a set of conditions are passed.

```
counter++;
var random_frame = (Math.floor(Math.random() * 50)) + 50
if (counter > random_frame) {
      var random = (Math.floor(Math.random() * 100))
      if (random >= 98) {
            bean2_right = !bean2_right;
            counter = 0;
      }
}
```

First, we generate a random_frame from 50-100. If the counter variable (started in the code above) is higher than the random_frame, then the code moves to the second conditional. This first gate is to ensure that the bean moves at least 50 frames in the same direction before flipping direction. Otherwise, as we discovered, because the game updates so quickly, regardless of how low the chances were of changing direction, the bean had a habit of staying in the same place and vibrating.

Launching 'Bean Bullets'

We toyed with a lot of different methods for creating the arc of the bullet. We tried a circle function and an inverted x-squared function, but eventually we settled on using part of a sine wave function. The biggest confusion we ran into was that the canvas element maps the origin in the top-left corner. So left to right is increasing, but so is up to down. It took us a little while to get our positives and negatives in the right place, but basically this is what it ended up looking like:

```
if (bullet.y < 401) {
    bullet.x = bullet.centerX +
    Math.cos(bullet.angle) * bullet.radius;
    bullet.y = bullet.centerY + Math.sin(bullet.angle) * bullet.radius;
    bullet.angle += bullet.speed / 40;
}
```

When spacebar is released, update() makes some quick calculations. The centerX is the x-coordinate halfway point between the final length of the shot and the bean's position. The radius is the actual distance between the halfway point and the bean's position. The angle varies between

0 and 180 depending on which bean is firing. All of these numbers of plugged into the function that calculates the x and y position of the bullet for each iteration. Together, they create a nice curve.

Character Lives (Health)

The game would've been over too quickly if it ended on one hit, so we gave each bean three lives. At first, our implementation was just a text counter (Health: 3) that decremented the number of lives from 3 to 0. Then we changed it to bean images that disappeared one by one on each hit. We positioned three clones of each bean image on top of the canvas, and made them visible on game start. When a bean is hit, a series of conditional statements control how many bean lives are now shown, depending on what is currently on the canvas. This code below is basically saying, "On hit, if Bean Life 1 is still visible, hide Bean 1. If Bean Life 2 is visible, hide Bean Life 2 (Bean Life 1 is still hidden). If Bean Life 3 is visible, hide Bean Life 3 and end the game."

```
 if (b2health_1.ready) {
        b2health_1.ready = false;
 } else if (b2health_2.ready) {
        b2health_2.ready = false;
 } else {
        b2health_3.ready = false;
         bean2Ready = false;
         gameOver = true;
}
```

Changing Bean Expressions

We essentially had a working and somewhat fun game after incorporating all the above. But we realized that the game was missing a key element: feedback for hitting or getting hit by the enemy. A reward and punishment system is a crucial component of game design, and although the beans had three lives that decremented, we felt some blood had to be shed.

Our first approach was to show a red container block over the bean characters and fade it out after a couple of seconds. When that didn't work for reasons we still can't fathom, we decided to create new bean images with hurt expressions and have them replace the original images on hit. It wasn't hard swapping out the images, as we just had to toggle the imageReady booleans. The trick was getting the hurt expression images to show for a few seconds before reverting to the normal images. We tried using the setTimer function to change the images back after 2 seconds, but the beans would either only flash their hurt expressions for a fraction of a millisecond (no amount of adding to that parameter increased the time) or disappear forever more. So we resorted to good ol' counters and incremented a 'hurtness' integer every time the screen updated. Once this counter reaches 100 (about 1 second), the original image comes back.

```
if (b1hurtReady) {
      context.drawImage(b1hurtImage, bean1.x, bean1.y);
       if (b1_hurtness > 100) {
            b1hurtReady = false;
            bean1Ready = true;
            b1_hurtness = 0;
       }
       b1_hurtness++;
}
```

V. Conclusions

We're pretty happy with the way BattleBeans turned out and feel that we can leave it in its current state and move onto other projects without shame. However, positive feedback from friends also tempts us to take the game to the next level. Either way, it's always there as a side project we can work on individually when we have the time and interest. We're also making it openly available on GitHub so that anyone else can improve on it.

Here are some of the next steps we would take (or suggest others to take) to continue developing it:

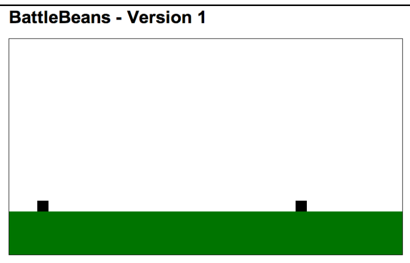
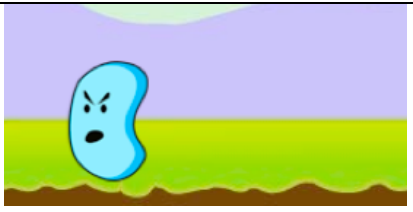
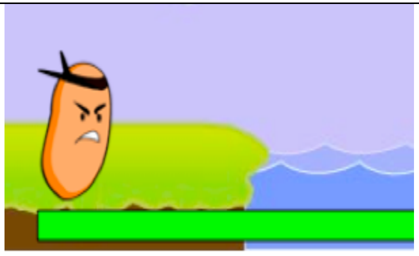
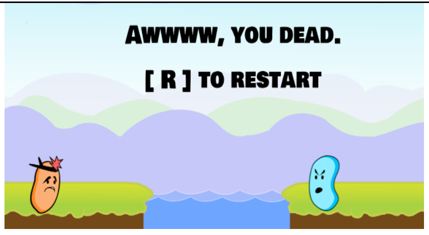
Basic Fixes/Priorities:
- scoreboard or way to accumulate points (so it's not just a one time game)
- display icon to turn music on/off
- pause game function
- being able to restart during game
- get bullet to display from beginning instead of midway

Add-Ons:
- 2 player mode
- graphic of what the bullet is being shot out of (other than their heads)
- different backgrounds to choose from
- different characters (who may have different abilities)
- levels: enemy gets faster or harder to hit
- drop bonuses randomly from the sky (extra lives, improved weapons, etc)
- landscape elevations

Our current JavaScript code is readable and acceptable (although there's plenty of room for improvement), but if we were to incorporate any of the Add-Ons, we would need to re-organize the code and maybe separate them into different files based on function.

Appendix

Our 'Expectations vs Reality' Timeline

	What We Planned To Accomplish	**What We Actually Accomplished**	**Pictures!**
Phase 1	Single player, flat landscape, two simple characters	Single player, flat landscape, two movable square characters, spacebar shoots a circle horizontally	 BattleBeans - Version 1
Phase 2	Introduce physics, varying elevations, ability to aim, power bars, movable players, turn-based	Improved graphics (landscape/bean characters), computer player with randomized movements, bullet stops on contact with target	
Phase 3	Computer player, improved graphics, game menu, scoring system	Horizontal power bars, bullet follows arc path (our version of 'physics'), computer player shoots back randomly, game quit function	
Phase 4	health points, variety of weapons/upgrades, single player campaign mode, top scores leaderboard	vertical power bars, health lives, game restart, splashscreen with menu, character expression changes, sounds/music	 AWWWW, YOU DEAD. [R] TO RESTART