Labs

The examples we will work though in class, and for assignments, will use the JavaScript programming language, embedded in web browsers. To make this easier, I have prepared a starter template and library in Codepen.io's online editor. This starter-kit will be used in all labs. The features provided by this kit are documented below.

Scripting with Codepen

Before anything:

- Make sure you have a recent version of Chrome or Firefox to work with, and you know how to open it's developer console.
- Then, make sure you have signed up for a (free) account on Codepen.

To create a new script, Fork this example

Alternatively, create a new pen on Codepen, open up the JS settings, and add paste the following into the External Scripts list: `https://alicelab.world/digm5950/al2019.js`

Tips

- **How do I view console output?**
  - Codepen includes a console, but I don't recommend it; the browser's built in developer consoles are far better. Instead, learn how to find the javascript console for your browser of choice:
    - Chrome: View > Developer > JavaScript Console
    - Firefox: Tools > Web Developer > Web Console
- **What happens if my pen crashes or goes into an infinite loop?**
  - On any page that displays a grid of Pens as live previews, or in the editor itself, you can disable the JavaScript with this query parameter at the end of the URL: `?turn_off_js=true`
  - When adding any kind of `for` loop, write the loop condition etc. in comments first, and uncomment once it is complete, so that you don't accidentally cause an infinite loop. Or, just turn live updates off in the codepen settings.
- **How do I get a local/offline copy of my work?**
  - You can export from Codepen to static HTML files using the Export (export as zip) button. You can also grab a local copy of the kit library by saving `https://alicelab.world/digm5950/al2019.js` and changing the script's `src` url in your html code to a local path.

Overview

Graphics rendering uses the HTML5 Canvas for 2D rendering. We assume the canvas is a world with coordinates ranging from (0,0) in the top-left corner to (1,1) in the bottom right corner -- i.e. it is square shaped with unit length.

Events

The kit provides an event-based callback system, in which you implement specially-named functions to handle specific events.

For simulation updates, implement a function called `update(dt)`:

```
function update(dt) {

}
```

The `dt` argument gives the delta time between updates (in seconds). Simulation can be paused and un-paused using the `Space` key.

To define how the canvas renders, implement a function called `draw(ctx)`.

```
function draw(ctx) {

}
```

All graphics code should go into this callback. The `ctx` argument is the Canvas 2d context which you can use for general 2D drawing. Rendering can toggle fullscreen mode using the `Escape` key.

For simulation restarts, implement a function called `reset()` This will be fired whenever you press the `Enter` key:

```
function reset() {

}
```

Additional callbacks exist to detect mouse/touch and keyboard interaction:

```
function mouse(event, point, id) {
    console.log(event, point, id);
}
```

- event == "down": a button was pressed / touch began
- event == "up": a button was pressed / touch ended
- event == "move": mouse/touch was moved (or dragged).
- the `point` argument is an array of two coordinates, in the range 0,0 to 1,1
- the `id` argument identifies which the button(s) pressed, or touch id in the case of multitouch

```
function key(event, key) {
    console.log(event, key);
}
```

- `event` == "down": a key was pressed.
- `event` == "up": a key was released.
- the `key` argument can be a string representation of the character pressed, such as "a", "2", "Z", ">" etc., or special strings such as "Shift", "Control", "Alt", "Meta", "Enter", "Backspace" etc.

Globals

The library provides a few extra global variables and functions that are frequently needed:

`now` is a global variable representing seconds since the script began

`random()` can be used to generate random numbers. Without an argument it returns rational numbers between 0 and 1; with a numeric argument it returns integers in the given range (e.g. useful for picking within an array).

`wrap()` applies a Euclidean modulo (remainder after division) in such a way that the result is always positive and without reflections.

`shuffle(arr)` randomly re-orders the array 'arr'. It modifies the array in-place, and also returns it.

```
random();         // a floating-point number between 0 and 1
random(6);        // an integer between 0 and 5
wrap(-1, 4);    // returns 3 (whereas -1 % 4 would return -1)
shuffle([a, b, c]); // returns [b, a, c] or [c, b, a] or [a, b, c] etc.
```

The `write()` function will output text above the main canvas. It may be more useful than calling `console.log()` in certain situations.

There are also a couple of global variables, used to measure time:

```
console.log(now);    // the time in seconds since the script started
console.log(dt);    // the delta time in seconds between each update()
```

2D drawing

The `draw` function's `ctx` argument is an HTML5 2D canvas drawing context, offering all the standard drawing capabilities of that API.

Alternatively, the `draw2D` namespace provides simpler interface for drawing 2D primitives.

Shapes

Basic shapes are as follows:

```
// many different ways to call this function using different arguments
// default center is 0,0, default diameter is 1:
draw2D.circle([center_x, center_y], diameter_x, diameter_y);
draw2D.circle([center_x, center_y], diameter);
draw2D.circle([center_x, center_y])
draw2D.circle(diameter_x, diameter_y);
draw2D.circle(diameter);
draw2D.circle()

draw2D.rect([center_x, center_y], diameter_x, diameter_y);
draw2D.rect([center_x, center_y], diameter);
draw2D.rect([center_x, center_y])
draw2D.rect(diameter_x, diameter_y);
draw2D.rect(diameter);
draw2D.rect()

// triangle points toward positive X direction
draw2D.triangle([center_x, center_y], diameter_x, diameter_y);
```

```
draw2D.triangle([center_x, center_y], diameter);
draw2D.triangle([center_x, center_y])
draw2D.triangle(diameter_x, diameter_y);
draw2D.triangle(diameter);
draw2D.triangle()
```

Colors

Graphics are drawn using whatever color is currently set. The current drawing colour can be set by assigning to `ctx.fillStyle`, or using these helpers:

```
draw2D.color("red")
draw2D.color("#ff3399")
draw2D.color(1, 1, 1, 0.5) // rgb+alpha
draw2D.color(1, 1, 1) // rgb
draw2D.color(1, 0.5) // greyscale+alpha
draw2D.color(0.5);  // greyscale

draw2D.hsl(0, 0.5, 0.5, 0.5) // hue, saturation, luma, alpha
draw2D.hsl(0, 0.5, 0.5) // hue, saturation, luma
draw2D.hsl(0, 0.5) // hue, alpha
draw2D.hsl(0.5);  // hue
```

The full list of named colors is here.

Transformations (coordinate spaces)

The coordinate system ranges from [0, 0] in the top left to [1, 1] in the bottom-right, but it is easy and useful to set up different coordinate systems, e.g. for drawing from an agent's point of view. `draw2D` uses a stack-based coordinate transform system. Push the context, apply transforms, then return back to the previous coordinate system by popping the context again:

```
// create a local coordinate system:
draw2D.push();
    draw2D.translate(x, y); // or a position vec2
    draw2D.rotate(angle_in_radians);  // or a direction vec2
    draw2D.scale(sizex, sizey); // or a size vec2

    // draw in local coordinate system
    //...

// return to global coordinate system:
draw2D.pop();
```

Most calls to draw2D can be chained together, since they return the `draw2D` object itself. Now typically, to move into an agent's coordinate system, operate in the order "translate, rotate, scale". So since most draw2D methods can also accept vec2 arguments, a common idiom is:

```
// push into agent's local coordinate system:
draw2D.push()
    .translate(agent.position)
    .rotate(agent.velocity)
    .scale(agent.size);

    // draw agent body -- the X axis is agent's forward direction
    draw2D.rect()
        .circle([0.5,  0.5], 0.5)
        .circle([0.5, -0.5], 0.5);

// done drawing agent
draw2D.pop();
```

field2D

The `field2D` type represents a dense grids of cells of floating point numbers (typically but not necessarily in the range of zero to one). You can create a field like this:

```
let field = new field2D(10);  // creates a 10 x 10 grid of cells
```

Typically we will render a field in the `draw()` callback by calling the field's draw method:

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

By default field cells will be zero, which looks black. You can get and set individual field cells this way:

```
let value = field.get(x, y);
field.set(value, x, y);
```

Note that if x or y is out of bounds for the field, it will wrap around from the other side. So you are always guaranteed it will return or set a value.

To set the value of all cells at once, omit the x and y:

```
// set all field cells to 1 (white):
field.set(1);
```

A more powerful feature of `field2D.set()` is that it can take a function instead of a number. When x and y are omitted, this function is called to update each cell in the field. The function receives as arguments the x and y position of the cell, so for example, this code initializes the field with a horizontal gradient:

```
field.set(function(x, y) {
    return x / this.width;
});
```

Note that `field.set` returns the field itself, so it can be chained.

More useful methods:

```
field.clear();         // set all field cells to zero, faster than field.set(0)
field.normalize();     // re-scales the field into a 0..1 range

field.clone();     // create a duplicate copy of the field
field.copy(another); // copy the values of `another` field (must have the same size)

field.min();    // returns the lowest cell value in the array
field.max();    // returns the highest cell value in the array
field.sum();    // adds up all cell values and returns the total

field.scale(n); // multiply all cells by n; AKA field.mul(n);
field.add(n); // add `n` to all cells
```

Normally fields render their cells with hard edges, but you can render the field more smoothly by setting:

```
field.smooth = true;
```

Normalized sampling

There are some methods for interpolated reading/writing/modifying fields. These methods use x and y indices in a normalized 0..1 floating-point range rather than 0..width or 0..height integer range:

```
// returns interpolated value at the normalized position x,y
let value = field.sample(x, y);
// or, using a vec2:
let value = field.sample(agent.position);

// adds v to a field at position x, y
// (interpolated addition to nearest four cells)
field.deposit(v, x, y);
// also accepts vec2
// a negative deposit is a debit (subtraction from field)
field.deposit(-v, agent.position);

// set the field at position x,y to value v
// (the four nearest cells will be interpolated)
field.update(v, x, y);
```

The field2D type also includes a diffusion method, which can be used to smoothly distribute values over time. It requires a second (previous) copy of the field to diffuse from. This method more than just a general blur -- it very accurately preserves mass before and after. So, for example, taking the `.sum()` of the input and output fields results in almost exactly the same quantity.

```
// field_previous is another field2D of equal dimensions
// diffusion_rate is a value between 0 and 1; a rate of 0 means no diffusion.
// accuracy is an optional integer for the number of diffusion steps; the default is 10.
field.diffuse(field_previous, diffusion_rate, accuracy);
```

There are also a couple of classic functional programming methods. The `map(function)` method applies a function to each cell of the field in turn. The function arguments will be the current value of the cell and the x and y position, and the return value should be the new value of the cell (or nil to indicate no change).

The `reduce(function, initial)` method is used to reduce a field to a single value, such as calculating the total of all cells. This value is defined by the `initial` argument, passed to the function for each cell, and updated by its return value. Easier to explain by example:

```
// multiply all cells by 2:
field.map(function(value, x, y) { return value * 2; });

// find the sum total of all cell values:
let total = field.reduce(function(sum, value, x, y) {
    return sum + value;
}, 0);
```

Multi-channel fields

Field cells are in fact 4-channel vectors, mapping to red, green, blue and alpha channels when rendered; however the methods described above are designed to simulate single-channel (greyscale) semantics.

Whereas `field.get` returns a single number (the red-channel value), `field.cell` returns the entire 4-plane vector as an array, which you can modify in-place to set a particular color:

```
// turn a cell red:
let cell = field.cell(x, y);
cell[0] = 1;
cell[1] = 0;
cell[2] = 0;
```

Alternatively, you can pass an array to `field.set`:

```
// turn a cell red:
field.set([1, 0, 0], x, y);
```

The normalized indexing and updating also supports multiple channels:

```
// to sample a specific channel (0, 1, 2 or 3):
field.sample(agent.position, channel);

// to update a single channel):
field.deposit(0.1, agent.position, channel);
// to update several channels:
field.deposit([1, 0.5, 0.1], agent.position);
```

Images

You can also load PNG images into a field2D.

```
field.loadImage("https://upload.wikimedia.org/wikipedia/commons/1/17/ArtificialFictionBrain.png");
```

Note: images will be resized to fit the field, not vice versa. Transparent PNGs are supported (the opacity information is in channel 4 of the field).

Note: images can take a little time to load. If you want something to happen only after the image is loaded, add a callback:

```
field.loadImage("https://upload.wikimedia.org/wikipedia/commons/1/17/ArtificialFictionBrain.png", function() {
    write("image is loaded!");
});
```

vec2

The `vec2` type gives us a useful abstraction of two-component vectors. It is simply an array of two numbers, `[x, y]`, but also has many useful methods available to it. Here are some ways of creating a vec2:

```
let v0 = new vec2(x, y);
let v1 = new vec2();     // x and y components default to zero

v0 = vec2.create(x, y); // equivalent to above
v1 = vec2.create();

let v2 = v0.clone();    // remember, v2 = v0 would not make a new copy

let v3 = vec2.random(); // a vector with unit length but random direction
let v4 = vec2.random(0.1); // as above, but length is 0.1

let v5 = vec2.fromPolar(len, angle_in_radians);
let v6 = vec2.fromPolar(angle_in_radians);    // length is 1
```

Getting some useful properties:

```
let d = v0.len();        // get the magnitude of the vector
let a = v0.angle();     // get the direction of the vector (in radians)
let d = v0.distance(v1);    // distance between two vectors
let a = v0.anglebetween(v1);    // angle between two vectors
let b = v0.equals(v1);  // true if both components are equal
let n = v0.dot(v1);        // dot product of two vectors (related to similarity)
```

There are many methods available to call on a vec2. Some basic setters:

```
v0.set(x, y);    // update a vector's values
v0.set(v1);        // update by copying from another vector

// changing the length (magnitude) of a vector:
v0.len(n);    // update a vector's length
v0.limit(n);    // shortens a vector to length n if it was longer
v0.normalize(); // set's a vector's length to 1

// changing the orientation of a vector:
v0.negate(); // reverses a vector
```

```
v0.angle(a); // set a vector's direction (in radians)
v0.rotate(a); // rotates a vector's direction (by radians)

// converting to whole number (integer) elements, e.g. for indexing a field2D by cell:
v0.floor(); // turns vector components into whole numbers by rounding down
v0.ceil(); // turns vector components into whole numbers by rounding up
v0.round(); // turns vector components into whole numbers by rounding to nearest
```

Almost all methods have both an in-place version and a standalone version. The in-place version is a method called on a vec2 object, which it will probably modify as a result.

The standalone version is a method called on `vec2` itself, and requires an argument for the result. Usefully, this argument doesn't need to be a `vec2` itself, it can simply be any array with two components.

```
// standalone:
// vout is modified; v0 and v1 are not changed
let v0 = vec2.create(1, 2);
let v1 = [3, 4];
vout = new vec2();
vec2.add(vout, v0, v1);     // vout is now [4, 6]

// in-place:
// v0 is modified; like the scalar equivalent s0 += s1
v0.add(v1);

// Since in-place methods return themselves, they can be chained:
v0.add(v1).sub(v1).mul(v1).div(v1); // etc.
```

Note that most method arguments will accept a number (i.e. a scalar), or an array, in place of a vector: So you can say `v0.pow(2)` rather than `v0.pow(new vec2(2, 2))`.

```
// These are all equivalent:
v0.pow(new vec2(2, 2));
v0.pow([2, 2]);
v0.pow(2);
```

Here are some basic algebraic methods, which operate on each component of a vector:

```
// component-wise math:
v0.add(v1);
v0.sub(v1);     // aka v0.subtract(v1)
v0.absdiff(v1); // the absolute difference between two vectors (always positive)
v0.mul(v1);     // aka v0.multiply(v1)
v0.div(v1);     // aka v0.divide(v1)
v0.pow(v1);        // raise v0 to the power of v1
```

There is a utility to perform linear interpolation between two vectors:

```
// a linear interpolation between v0 and v1
// if t == 0, result is v0;
// if t == 1, result is v1;
// if t == 0.5, result is the average of v0 and v1;
// etc. for other values of t
vec2.mix(vout, v0, v1, t);
```

There are also several useful methods for keeping vectors within bounds:

```
// max retains the values that are closer to Infinity
vec2.max(vout, v0, v1);
// min retains the values that are closer to -Infinity
vec2.min(vout, v0, v1);

// lesser retains the values that are closer to zero
vec2.lesser(vout, v0, v1);
// greater retains the values that are further from zero
vec2.greater(vout, v0, v1);

// aka clip: keeps v0 within the bounds of vlo and vhi
v0.clamp(vlo, vhi);

// like wrap(), it gives the remainder after division
// it uses Euclidean modulo, which handles negative numbers well for toroidal space
v0.wrap(d);

// applies wrap in a relative range, up to +d/2 and down to -d/2
// this can be useful e.g. for calculating the shortest distance between two points in toroidal space
vec2.relativewrap(vout, v0, v1);
```

hashspace2D

In general, a naive but effective approach to detect nearby objects is to iterate over all objects and compute the relative distances. For example:

```
for (let a of agents) {
    for (let n of agents) {
        let rel = vec2.distance(a.pos, n.pos);
        if (rel < (a.size + n.size)) {
            // they are overlapping
        }
    }
}
```

This approach, while effective, is a little inefficient. The `hashspace2D` type offers a more efficient way of detecting neighbours:

```
let space = new hashspace2D();

space.clear(); // remove all objects from the space

// hashspace2D keeps track of objects that have a position and size
// (with .pos and .size properties)
```

```
let obj = {
    pos: [0.5, 0.5],
    size: 0.1,
};

// add an object to the space:
space.insert(obj);
// move an object in the space:
space.update(obj);
// remove an object from the space:
space.remove(obj);

// search for any other objects within 0.1 distance of "obj"
// returns an array
let near = space.search(obj, 0.1);

// search for any other objects within 0.1 distance of the world center
// returns an array
let near = space.search([0.5, 0.5], 0.1);
```

JS Result EDIT ON

Run Pen

Resources 1× 0.5× 0.25× Rerun

Example scripts

The in-class scripts and examples will all be collected here: https://codepen.io/collection/nRZQOO?grid_type=list

2020