Fabric.JS

Working with events

Why events

Fabric.js offers hard coded interactions that are triggered by user interactions with the canvas. During those actions the developer may want to react or run extra code, or even just update the state of the application.

Simple examples are:

- When an object has been selected
- When an object is being hovered
- When a transformation is finished

Events have been always part of fabric.js, way before the custom controls api and selection callbacks were introduced. As a result events have also been used to modify standard fabric.js behaviours creating complex code flows.

When to use events

Try to do *not* depend too much from events, is not a great pattern outisde mouse and selections. At the time i joined the project events were already there and with time there have been additions but never cleanups. Try to think in this way: In Fabric.sj events are meant to alert you of things happening that you can't reach because are part of code you didn't write. Events aren't a message system to activate or connect together functionalities of your app.

There is an example i try to keep in mind when i decide to use or not use an event, the 'object:added' event. The 'object:added' fires when an object is added to the canvas or is

```
canvas.on('object:added', function(e) {
  canvas.center(e.target);
  canvas.setActiveObject(e.target);
});
```

Now from any part of your application that adds an object the event will fire and this code will run. It seems fine, but on the other side if you think that in order to add an object you need to write code for that to happen, you may also just create a function that adds an object, center and selects it and then just call that function instead of calling `canvas.add`.

In general if you need to write code for something to happen events from that something are unnecessary.

The custom controls api made most of the transform event unnecessary as well. If you have to constrain scaling or rotation it will be easier and more performant to do so in the control action rather than correct the behavior after the fact in the event. Transform events run after the transformation already happened.

Events around the selection actions are good to inform your UI framework that something has happened, but for example if you want to avoid selecting an object based on some condition it is better to use the onSelect and onDeselect callbacks on the object that are made to influence the operation.

How events work

Both the BaseFabricObject class and the StaticCanvas inherit from an Observable class. This class expose 4 methods, `on`, `off`, `once` and `fire`.

The `on`, `off`, `once` are used to register and unregister event listeners:

```
const handler = function() {
  console.log('circle has been clicked');
}
// to register an event listener
const disposer = myCircle.on('mousedown', handler);

// to unregister an event listener call off
myCircle.off('mousedown', handler);
// or use the return value of on
dispose();
```

Be careful: The `off` method can be called with only the event name, in this case all the listeners for that event will be removed from the instance. This is bad since fabric.js does not have protected events and it is using events events for some functionalities. For example removing all the mousedown events from a text instance will break text editing. This pattern and the more radical one `myObject.off()` are deprecated and should be avoided.

The disposer is a function that when called will remove the event listener, this may be easier to use than keeping a reference to the handler function used with `on` as an alternative.

The handlers are called bound on the instance, so in the handler funcion the `this` is either the fabric object or the fabric canvas. Fat arrow functions can't be bound to anyhting else than the context in which they were created, so if you don't want unexpected side effects **use functions for event handers**.

Fabric.js calls `fire` to trigger event listeners, the listeners that are hosted on the instance are then called bound in order of registration and with a data object as first and only argument.

Events are fired twice, once on the object involved and once on the canvas, this allow a bit of freedom in writing your event listeners. The fire order depends on the implementation of the particular event and is not a specification to take in account. While firing twice the event is the same, with the same data object. The canvas event is enriched with an additional property referencing the `target` of the event.

- `e` the original mouse event
- `scenePoint` the point where the event happened in the canvas coordinate system
- `viewportPoint` the point where the event happened in the viewport coordinate system
- `target` the object that was hit by the event, possibly undefimed
- `subTargets` an array of objects that were hit by the event, possibly empty, if the target has children
- `transform` the current ongoing transform action (if any, like scaling for example)

Events list

There isn't an hand curated even list with descriptions and data passed in the argument. One way to get to know the events is to look at the event demo here: Events inspector or to use a type enabled ide and look at auto completition for the on method.

Custom events

You can fire custom events by using the `fire` method and passing the event name and the data you want, or no data at all.

To keep your code typehinted, you can also extend the event list:

```
// declare the events for typescript
declare module "fabric" {
  interface CanvasEvents {
      'custom:event': Partial<TEvent> & {
          target: FabricObject;
          anydata: string;
      };
  }
}
```