Renderers

Edit New Page

Jump to bottom

At-Low edited this page on May 11 · 16 revisions

Sigma is no more an all-in graph renderer based on the Canvas DOM element, but now it is a controller, that can be bound to multiple renderers.

Currently, in the trunk, sigma provides two different renderers:

- a Canvas based renderer, pretty simple to use and to hack.
- a WebGL based renderer, way more powerful, but using a different data pipe, that is way harder to hack.

How it works

When the user displays a graph, here is what happens to the nodes coordinates:

1. The `sigma.middlewares.rescale` function will make the graph fit into the screen.
2. Each camera will apply its position to each node.
3. Each renderer will take the coordinates of its bound camera to render the nodes.

With the Canvas renderer, to keep all these steps independent from each other, each of these routines takes a **prefix** as input and sets the coordinates with a new output. This makes possible to have multiple cameras running at the same time in the same sigma instance.

But the WebGL renderer **uses matrices** instead of applying a function to every node at each step. Indeed, the main advantage of using WebGL is that the graphic card is much more powerful than the CPU to do matrices computations. So, each step is modelized as a matrix, and the product of all those matrices will be applied to each node at the final rendering step.

So, to sum up:

- With the Canvas renderer, the nodes will go through different functions to determine the final coordinates. The advantage is that **you can do whatever** you want with functions, but it will be expensive, since the nodes array must be looped several times.
- With the WebGL renderer, the nodes are never looped, and all the modifications to apply are synthetized in a unique matrix. The advantage is that **the nodes are never looped or modified** and **everything is**

computed on the graphics card. But the biggest disadvantage is that you can only use matrices to modify nodes coordinates.

The Canvas renderer

The Canvas renderer is pretty similar to the one used in the old version of sigma. Basically, it renders edges, nodes and labels on three different layers (so, three different Canvas elements), plus another layer that catches the mouse events and displays the hovered nodes.

It renders **only nodes that are on screen**, and **edges that have at least one extremity on screen**. To avoid having nodes disappearing from screen as soon as their center leaves the screen, a margin is actually added to the screen to find what to display. This explains why you might see edges disappearing from the screen while zooming, for instance.

To draw a node, the Canvas renderer will get its `type` attribute, and check in the `sigma.canvas.nodes` package if a nodes renderer is associated. It will use it if it exists, and `sigma.canvas.nodes.def` if not or if the node has actually no `type` attribute.

It works exactly the same for labels and edges with the packages `sigma.nodes.edges` and `sigma.nodes.labels`, respectively.

So, if you want to develop a custom nodes renderer, for instance one that displays the node as a square, you can just write a plugin that sets the function `sigma.canvas.nodes.squares`. The easiest way for you would be to get started on Canvas drawing techniques, and copy the code of the default nodes renderer: `sigma.canvas.nodes.def` before changing the code to make it actually draw a square.

Here is how the new nodes renderer's code would look like:

```
sigma.canvas.nodes.square = function(node, context, settings) {
  var prefix = settings('prefix') || '',
      size = node[prefix + 'size'];

  context.fillStyle = node.color || settings('defaultNodeColor');
  context.beginPath();
  context.rect(
    node[prefix + 'x'] - size,
    node[prefix + 'y'] - size,
    size * 2,
    size * 2
  );

  context.closePath();
  context.fill();
};
```

At this point, if the previous script has been loaded into your application, any node with the `"square"` value for its `type` attribute would be displayed with this renderer, so as a square.

It will work exactly the same to develop custom edges or labels renderers.

The WebGL renderer

Currently, the WebGL renderer draws nodes and edges on two different layers using WebGL. It will deal with labels, mouse and hovered nodes display with simple Canvas, exactly as the Canvas renderer does. The reason is that displaying texts with WebGL is hard.

The nodes and edges renderers are respectively stored in the packages `sigma.webgl.nodes` and `sigma.webgl.edges`, so you can add your own as plugins easily, as with the Canvas renderer. The difference is that a WebGL node or edge renderer is not a simple function: Check the end of `sigma.renderers.webgl.js` for more information about how it works.

Here is the precise description of the whole rendering process with WebGL:

1. When the `refresh` method of the sigma instance is called, it will call WebGL renderer's `process` method.
2. This method will generate for each nodes renderer a typed array describing each node with the related type, through the nodes renderer own `addNode` function. Each renderer requires its own typed array because it has its own format, that is recognized only by its own shaders. The same process is applied to edges.
3. Then, the `refresh` method of the WebGL renderer will be called.
4. Each nodes renderer will have its own `render` method called with the related typed array. The same process will be applied to edges.
5. For the labels, the quadtree will tell which nodes must be displayed.

If you need to know more about how it works, the best thing to do is to go read the code. But be conscious that **it is way more painful to get started with WebGL than with Canvas**.

The default renderer

Sigma also provides a default renderer `sigma.renderers.def`, that detects if the browser supports WebGL, references the WebGL renderer if yes, or the Canvas renderer else.

If you want to specifically use the Canvas renderer when instanciating sigma, you must describe precisely the renderers (check the core API documentation). Here is an example:

```
var s = new sigma({
  renderers: [
    {
      container: document.getElementById('myContainer'),
      type: 'canvas' // sigma.renderers.canvas works as well
    }
  ]
});
```

Building a custom renderer

As said earlier, it is possible to plug your own custom renderers to sigma. Basically, a renderer is a class that must have a **render** method, and a **width** and a **height** attributes. These size attributes are required for the `rescale` routine that makes the graph fit to the screen.

Also, it is possible to add a `process` method, if the renderer needs to "prepare" for rendering. For instance, an SVG renderer might spawn DOM elements on the `process` step, and just update their properties on the `render` step. Also, as seen earlier, the WebGL renderer regenerates its typed arrays when the graph has been modified in its `process` method.

Check the existing renderers source code to know more.