A First Graph Library for Babylon.JS
Final Report

John Moran
jmoran@eagles.bridgewater.edu

R. Eric McGregor, Ph.D.
rmcgregor@bridgewater.edu

Introduction

An interesting object that can be displayed in the browser is the graph. Here, graphs are sets of nodes that are connected via edges. Each node represents an entity and an edge between two entities, often depicted as a line, represents the existence of a binary relationship between the two entities.

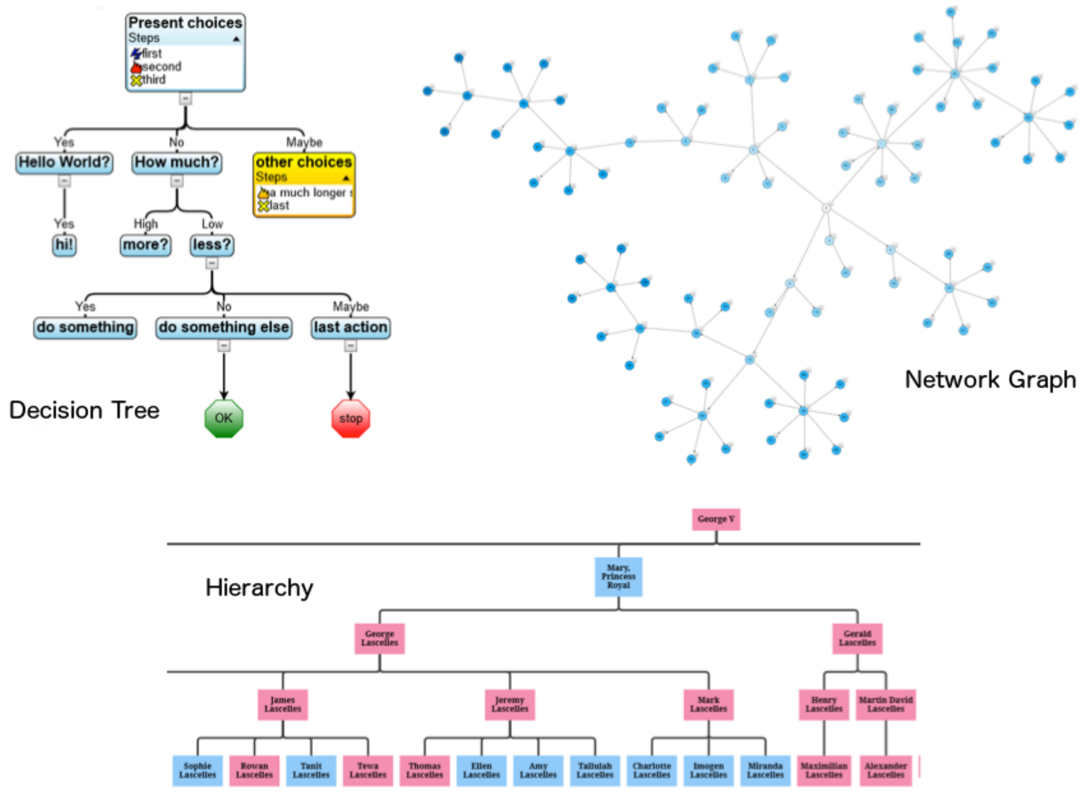


Figure 1: A collection of graphs created using VivaGraph

These 2D graphs are very useful models when the edges don't overlap; however, when they do, understanding the relationships between entities becomes more difficult as shown in Figure 2.

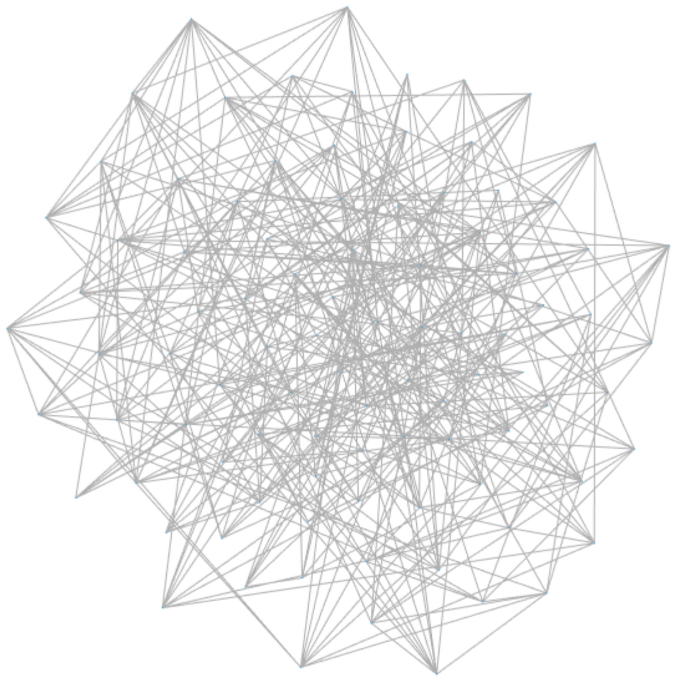


Figure 2: A graph with 93 nodes and 692 edges.

Graphs can be dynamically created and viewed in a browser using the JavaScript programming language. Here, the web developer can code an application to allow a user to pan left and right and zoom in and out to focus on different areas of the graph. This permits very large models with hundreds of nodes and hundreds of edges to be displayed. Some of the JavaScript libraries that allow a programmer to create graphs dynamically in a 2D environment include D3.js and VivaGraph [1,2]. One drawback however is that the graphs are flat and stationary.

Recently, technology has been developed that allows the rendering of 3D scenes in the web browser. Prior to this advancement, 3D environments were restricted to gaming consoles and high performance computers. Today there are two JavaScript frameworks, Three.js and BabylonJS [3.4] that allow information to be displayed in the browser in 3D. Here, a 3D scene is created that can include 3D objects with textures, lighting, shadows, particle systems that simulate fire and rain, cameras, and physics effects such as gravity. Users, with their arrow keys, can navigate the space and with the click of a mouse can rotate and interact with objects in that space.

In this project we have created a JavaScript graph library called YAGL (pronouned 'yæ gəl) for the Babylon.js 3D framework. This library allows programmers to easily create and manipulate 3-dimensional graphs in a 3D Babylon scene and perform a few algorithms on the graph.

YAGL Files

In the YAGL library a graph is specified in a YAGL object or file. A YAGL object or file is simply a JSON object or file that defines the vertices (a.k.a. nodes) and edges of a graph. An example is give below.

```
{
   "description": "example",
   "graphicsFramework":"Babylonjs",
   "layout": "Force Directed",
   "vertices":[
      {"id":1, "position":[0,0,0],
         "mesh":{
            "meshName":"Suzanne",
            "rootUrl":"http://demo.yagljs.com/assets/",
            "sceneFilename":"suzanne.babylon"
         }
      },
      {"id":2, "position":[0,0,5]},
      {"id":3, "position":[5,0,5]},
      {"id":4, "position":[5,0,0],
         "mesh": {
            "meshName":"Cube",
            "rootUrl":"http://demo.yagljs.com/assets/",
            "sceneFilename":"cube.babylon"
         }
      },
      {"id":5, "position":[0,5,0]}
   ],
   "edges":[
      {"id":1, "v1":1, "v2":2},
      {"id":2, "v1":2, "v2":3},
      {"id":3, "v1":3, "v2":4},
      {"id":4, "v1":4, "v2":1}
   ]
}
```

The "description", "graphicsFramework" and "layout" properties are optional at this time. By default the library supports only the BabylonJS graphics framework and an automatic force-directed layout. The "vertexMesh" property defines the default mesh to be used when a vertex is specified without a mesh property. Both the "vertexMesh" property and the "mesh" property specify a mesh name in a Babylon scene file. These meshes are imported using Babylon's AssetsManager.addMeshTask() method.

The "vertices" property holds an array of vertex objects, each having "id", "position" and optional "data" and "mesh" properties. The "id" is an integer that can be referenced when specifying an edge. The "position" property specifies an x, y and z coordinate for the vertex in an array. If a position is not specified for a vertex the LayoutManager will assign the vertex a position. The "data" property can be used to store information associated with a particular vertex which can be set and access by the web application.

Following the "vertices" property is the "edges" property that consists of an array of edge objects. Each edge object contains "id", "v1" and "v2" properties. The "id" property is an integer and the "v1" and "v2" properties refer to vertex ids.

The YAGL Classes

The YAGL library has 4 main classes: GraphBuilder, Graph, GraphicsManager, and LayoutManager. The GraphBuilder class is responsible for parsing the YAGL object or file and for creating and building a model of the graph via the Graph class. A Graph object holds the model of the graph and additional data that needs to be stored in the nodes of the graph. When a Graph object is first created by the GraphBuilder, the Graph constructor creates a GraphicsManager object. Each time a vertex or edge is added to the model by the GraphBuilder it calls a Graph method, which calls upon a GraphicsManager method to return a 3D Mesh object. This mesh object is then stored in the graph model and displayed in the scene. When initialized, the GraphicsManager creates a LayoutManager object that, when necessary, helps find a reasonable position for a mesh in a scene using a force-directed algorithm [5] if a position is not specified explicitly.

Creating a YAGL Graph

To create a graph from a YAGL file one simply creates a GraphBuilder object (passing to the constructor the Babylon scene object) and then calls buildUsingJSONFIle() (passing it the URL to the file) as show below.

```
var builder = new YAGL.GraphBuilder(scene);
builder.buildUsingJSONFile(url_of_yagl_file);
```

Similarly, you can create a YAGL graph from a YAGL object.

```
var builder = new YAGL.GraphBuilder(scene);
builder.buildUsingJSONObj(your_object);
```

You can also incrementally build a graph using the GraphBuilder's addVertices() and addEdges() methods.

YAGL Algorithms

Once a graph is constructed a number of algorithms can be performed on the graph. Below are brief descriptions of the algorithmic methods in the Graph class.

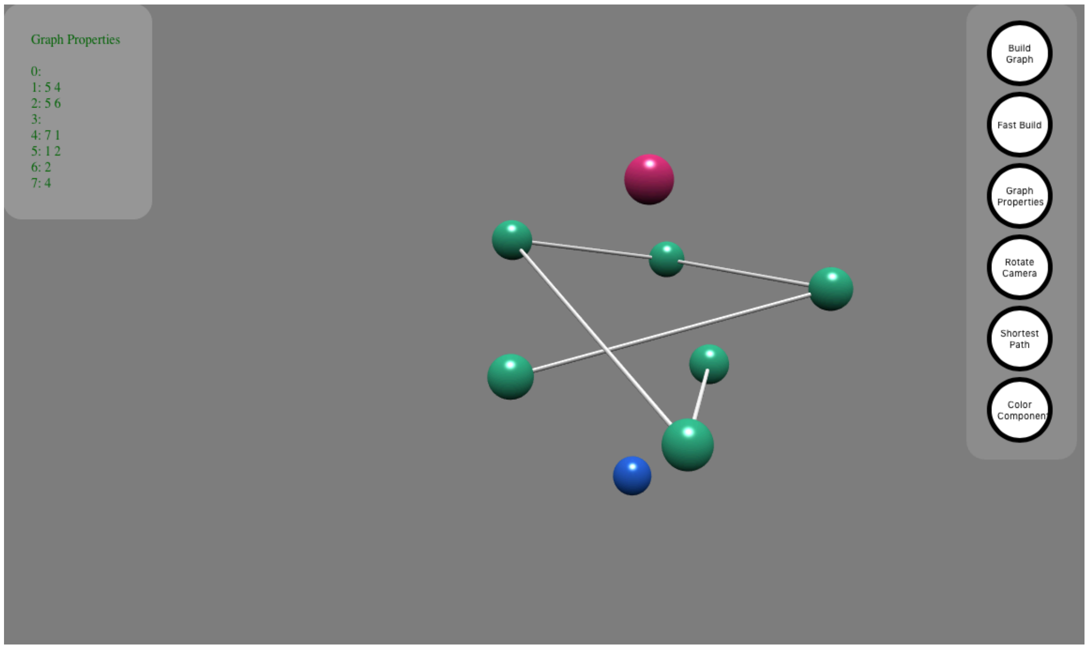
- isConnected(): This method returns true if every pair of vertices are connected by a path in the graph. This is implemented via a Union-Find algorithm.

- BFSearch(rootVid, searchVid): This method searches a graph using a Breadth-First search starting at rootVid and continues until it finds searchVid in which case it returns the vertex, or it has traversed all of the nodes connected to rootVid in which case it returns null.

- getPath(vid1, vid2): This method returns an array containing a shortest path from vid1 to vid2.

Demonstration Website

We've also created a few web applications that demonstrate some of the uses of YAGL. The first basic Scene application (http://demo.yagljs.com/basic/basicScene.html) displays a vertical row of buttons on the right and a dialog box on the left. The graph is displayed in the center of the page. The buttons allow the user to build a graph (either randomly or from predefined files), adjust the build speed, toggle on and off graph information in the dialog box, freeze the scene camera, compute the shortest path between two nodes and color the different components in the graph with different colors.



```
Graph Properties

0:
1: 5 4
2: 5 6
3:
4: 7 1
5: 1 2
6: 2
7: 4
```

Figure 3: A randomly generated YAGL graph with 3 components

The second demonstration web application (http://demo.yagljs.com/fb/facebookScene.html) requires the user to have a Facebook account and to log into their account. The app uses the Facebook Graph API to create a graph (with cube vertices) of the user's friends who have visited the demonstration page. Each node of the graph is clickable and when clicked displays a link to the friend's Facebook timeline.

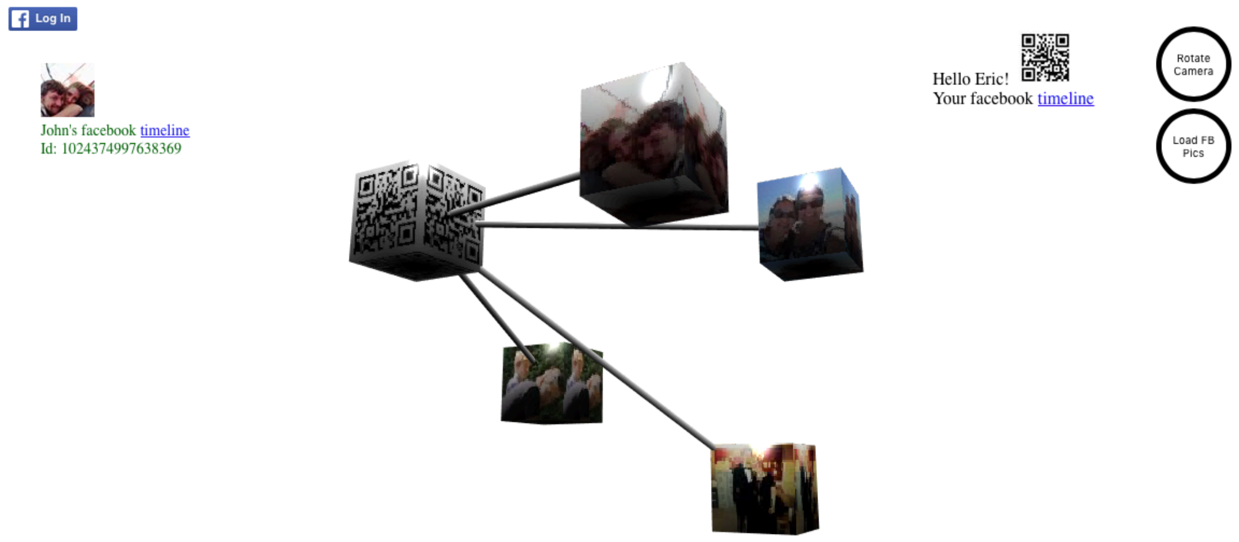


Figure 4: A YAGL graph created using a user's Facebook graph

Source Code

The source code for YAGL and the demonstration applications can be found at https://github.com/erimcg/YAGL. We have submitted a request for the source code to be open sourced and are awaiting approval from the college administration.

Future Work

In the future we'd like to add subclasses of the Graph class such as Tree, DirectedGraph, WeightedGraph and NetworkFlow. Along with these we'd like to implement more algorithms like HAS CYCLE, MINIMUM SPANNING TREE, and MAXIMUM FLOW. We'd also like to develop a demonstration web application that allows a user to create a graph using a mouse and a pallet of options (like a painter) and add the ability for users to export graphs to .yagl files.

Bibliography

1. Bostock, Mike (2016) D3.js [Source code]. Available at https://github.com/mbostock/d3 (Accessed April 16, 2016)
2. Kashcha, Andrei (2016) VivaGraphJS [Source code]. Available at https://github.com/anvaka/VivaGraphJS (Accessed April 16, 2016)
3. Cabello, Ricardo (2016) Three.js [Source code]. Available at https://github.com/mrdoob/three.js/ (Accessed April 16, 2016)
4. Catuhe, David (2016) BabylonJS [Source code]. Available at https://github.com/BabylonJS (Accessed April 16, 2016)
5. Hotson, Dennis (2013) Springy [Source code]. Available at https://github.com/dhotson/springy (Accessed September 18, 2016)