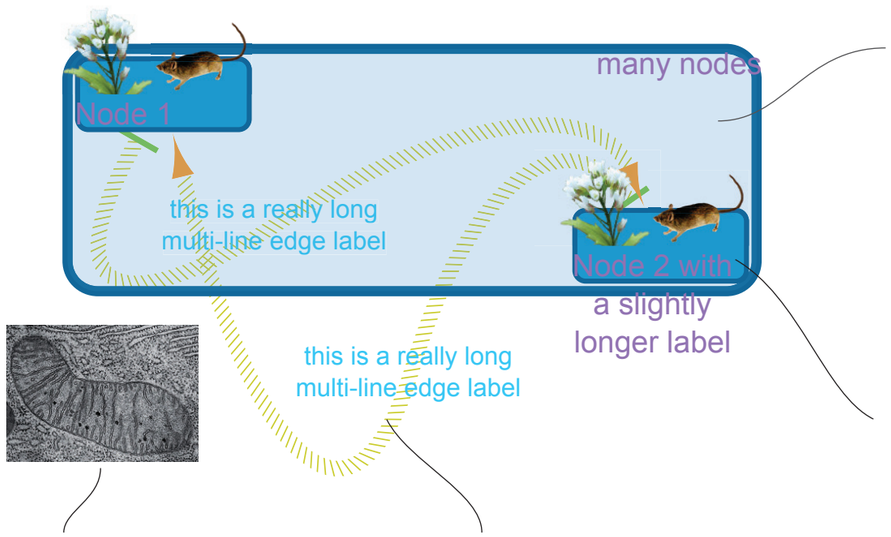
Zugzwang OpenGL renderer, more details

Ding can draw many things

Let's keep it that way. Node drawing occurs **completely in Java**, reusing Ding's procedures. Maybe over time a way can be found to share this code between both renderers to facilitate maintenance. Virtually all of Ding's drawing capabilities can be replicated using the OpenGL-enabled architecture.
The only part that should be done differently in OpenGL are visual properties describing **selected nodes/edges**. Giving the entire texture a **color tint** in a GPU shader is easier than redrawing the texture in Java to update the color of specific parts.



Compound node/group

At the expense of visual customization this could be made a **convex hull** of its child nodes, possibly interacting with the **layout algorithms**.
A convex hull would be a (flat in 2D) volume defined by a set of triangles, with variable color and transparency. A **silhouette** with variable color/thickness could be extracted separately in hardware or software.
Compound nodes of the same visual complexity as general nodes are expensive in terms of texture memory because they are big.

Node

When drawn completely, has 2 textures on the GPU:
–**Node label**
–**Node shape**, i. e. all the graphics drawn onto a single rectangle, excluding text

It also has a **binary mask** on the CPU for hit tests; can be compressed to 1 bit/px, or even less (e. g. run length compression) at some computational expense. This mask does not necessarily have the same dimensions as node shape/test. Cytoscape currently doesn't do hit tests outside the core rectangle. Not completely drawn nodes are fine with simple parametric description (i. e. rectangle, circle).

To OpenGL, all nodes are textured 2D sprites facing the camera. However, an option is to have an additional 3-channel texture that provides **per-pixel shape normals** in the first two channels, and **specular hightlight intensity** in the third channel. Nodes with this texture are rendered using a shader that considers the global light sources and these normals to **shade the node dynamically**.

Annotations

Are just 2D now and have no functionality. They should be much more.

The renderer must provide access to its glContext and the current view & projection matrices. However, not everyone will want to render through OpenGL. A volumetric raytracer is easier done with OpenCL, future technologies might provide even more possibilities.

These data are needed for full integration:
–**View & projection** matrices
–**glContext**
–**Depth buffer**, and information about near/far clipping planes in case it is normalized
–**RGBA buffer**, although Alpha would lose its purpose because the depth buffer doesn't allow transparency

In return, the object provides a **bounding box** which it will not spatially exceed. This box is used for the first level of **hit testing**. If the box is hit, the ray is given to the object for further evaluation.

Edges

Divided into 4 parts:
–Elongated rectangle, possibly tesselated into a smooth strip of small rectangles, carying a tiled texture or just solid color to represent the **edge line**.
–Small rectangle for **source arrow shape**.
–Small rectangle for **target arrow shape**.
–Arbitrarily sized rectangle for **edge label**.

Edge textures are very limited compared to nodes and can all be prerendered. However, different resolutions are required and simple downsampling might give blurry textures; thus, manually prerender crisp textures for multiple resolutions.

Other than nodes which are always parallel to the view plane, edges can start and finish at different depth levels, showing **perspective distortion**.

Camera and light controls

Use the space currently occupied by bird's eye view for extensive camera controls. Current scene rotation can be represented by something like the **ViewCube** from 3ds max. If directional **light sources** exist, they can be included in this schematic drawing as well, with the option to change the light direction by dragging the icons (similar to Chimera). The cube is probably easier to handle entirely in Java.

In addition to the ViewCube's functionality, buttons to **quickly zoom out** to see the whole scene and return to the **previous zoom level** are necessary to replicate bird's eye view. **Undo/redo** buttons for view changes are essential.

Instead of just one "home" view setting, an arbitrary number of **view configurations** can be **stored and retrieved** to prevent the user from getting lost.

For camera control directly from the viewport, this scheme could work:
–Mouse wheel drag = **pan** view
–Ctrl + wheel drag = **rotate** view
–Ctrl + wheel drag close to viewport border = **roll** view
–Mouse wheel scroll = **zoom** view
Additionally, **multi-touch** gestures can be implemented later, as they work really well in Chimera on OS X.

View rotation should only affect **pitch** and **yaw**, not roll. **Roll** is controlled separately. No "move mouse in circles to roll the camera" effect like in Chimera.

Level of detail

Technically, this should not be necessary – the GPU can probably push more nodes to the screen than Cytoscape's memory footprint would allow. However, things like text labels become useless at low zoom levels and become visual clutter. Either offer the user a button to quickly turn off elements globally, or employ a **heuristic based on node scale**. The latter would be more important in 3D, where nodes can have vastly different scale values due to perspective.

Default node appearance

A node/edge that has not been drawn yet, or does not fit into texture memory (and cannot displace another one with lower priority) must still look like a node. Maintain a pre-rendered default shape that can be used at any time. Switching to a more detailed shape is then a matter of changing texture IDs. For obvious reasons, labels are unique and cannot have a generic representation.

Hit tests, selection

Very simple strategy for all **nodes**: transform center position and bounding sphere radius into projection space; do 2D distance evaluation between cursor position and all bounding circles; if hit, check against sprite rectangle; if hit, calculate position within the binary mask and check value; if hit, add node to candidates list; sort candidates by depth, finally select the one closest to the camera.

Edges are harder: transform into projection space, check hit with bounding circle or rectangle (circle can get disproportionately big); if hit, tesselate the edge to the same level as on the GPU, using the same scheme; check distance between pointer position and each segment; edges can have perspective distortion in 3D, consider this; if hit, add to list; sort list by distance, pick the one closest to the camera.

General objects (like annotations): perform tests with bounding boxes; give ray to object for further evaluation; objects returns hit/miss and (approximate) hit position; if multiple hits, sort positions by distance, pick the one closest to the camera.

These protocols can be adapted to **selection rectangles** as well.

OpenGL features used

OpenGL 4 offers several techniques that can help to keep the communication overhead between CPU and GPU to a minimum for this task:

Instancing: all nodes have exactly the same geometry – a rectangle. The rectangle only needs to be defined once. The same vertex data can then be reused by merely looking up 12–13 values per node, or 72–88 byte: **shape position** xyz, **shape size** xy, **label position** xyz, **label size** xy, 2–3 **texture IDs**. These values reside in separate buffers and are only updated on changes, the most frequent being position (node movement, but not camera movement).

Geometry shaders: the ability to create new triangles directly on the GPU allows to reduce a textured edge ribbon to a simple line; each segment then spawns 2 triangles on the GPU. Additional per-edge parameters are **thickness**, physical **length of the tiled texture**, and **texture ID**. Similar idea for source and target **arrowheads**, they can be represented by single line segments.

Tessellation shaders: before the edge lines are extruded to ribbons in a geometry shader, they can be tessellated using the Catmull-Clark scheme to better approximate the curve.

Bindless textures: bindless textures can be referenced directly through a 64 bit handle, instead of binding the texture to a sampler using the API each time it is used. The overhead for having more textures is negligible, enabling hundreds of thousands of individual textures instead of one big, wasteful texture atlas.