Introduction

The aim of this project is to develop a collection of applications which can render custom graphics. Inspiration is taken from the VizRT suite of applications. Currently this collection consists of Chroma Viz, Hub, Engine and Artist. Chroma Viz, Hub and Artist are built in Golang and are contained in the [Chroma Viz](https://github.com/jchilds0/chroma-viz) repository on Github. Chroma Engine is built in C and is contained in the [Chroma Engine](https://github.com/jchilds0/chroma-engine) repository. <video width="720" controls>

```
    <source src="https://github.com/jchilds0/chroma-viz/raw/main/data/demo.m
```

</video>

Chroma Viz

Chroma Viz manages templates at a high level, and issues commands to Chroma Engine. On startup, Chroma Viz requests the templates from Chroma Hub. Chroma Hub collects the template IDs of all templates in the hub and sends this list to Chroma Viz. <img src="/assets/chroma-graphics/templates.png" alt="Templates"> Pages can be created from templates, by double clicking on the template in the Templates tab. Chroma Viz will then send a request to Chroma Hub for this template. Chroma Hub implements a REST API for accessing assets such as templates. Pages form shows, which can be saved to/loaded from disk for future use. <img src="/assets/chroma-graphics/show.png" alt="Templates"> Chroma Viz can connect to any number of Chroma Engine instances, using tcp sockets to communicate. Each connection is either a Engine or Preview connection. When the user opens a page to edit by double clicking on the page, or by saving changes made to the page with the save button, the page is sent to all connections with the preview type. Chroma Engine provides a C library which can create a GtkGLRender widget, and using cgo, Chroma Viz creates a Chroma Engine preview window. Chroma Viz sends pages to layer 0 of the preview window, so pages switch as expected which changing between pages. The actions at the top of the editor panel, $ \texttt{Take On, Continue} $ and $ \texttt{Take Off} $, send pages to connected Chroma Engine instances with the Engine type.

Take On animates from Keyframe 1 to Keyframe 2.

Continue runs from the current Keyframe to the next Keyframe.

Take Off runs from the second last Keyframe to the last Keyframe.

Chroma Viz encodes the attributes of a page and sends it to Chroma Engine. Chroma Engine parses the message and updates the templates attributes before rendering the graphic. <img src="/assets/chroma-graphics/chroma-viz.png" alt="Chroma Viz">

Chroma Engine

Chroma Engine renders graphics requests from Chroma Viz. At its core, Chroma Engine creates a GtkGLRender widget which renders graphics. We compile both a library, which is used by Chroma Viz to create a preview window, and a binary, which creates a standalone GTK application which only contains the GtkGLRender. On startup, Chroma Engine connects to Chroma Hub and requests all templates in the Hub. This is done so Chroma Engine can build its own database containing each template that could be received from Chroma Viz. This has the added cost of needing to allocate resources for each template, but the benefit is we don't need to allocate memory at run time when we receive a graphics request. A middle ground between these two options would be allowing the user to load a subset of templates currently in use, and requesting any new templates on the fly from Chroma Hub as needed. Chroma Engine renders graphics using OpenGL. To render a graphic, first Chroma Engine recieves a string of data from a Chroma Viz instance. A simple name-attribute format name=attr#is sufficient for our purposes. The string contains a header with the format version, layer, template id, and action. Then each geometry, specified by an integer, followed by a list of attributes for the geometry. As we parse the string, we set the values for each geometry of the target template. Chroma Engine features geometry masking. Each geometry has a masking attribute, and if set, for each pixel of the geometry we check if every parent geometry also has a pixel at this point. To achieve this we utilise OpenGL stencil buffers, keeping a buffer for each parent and then drawing the geometry where we can pass through all buffers. GTK restrictions mean we can only have 8 stencil buffers, restricting geometry tree depth to a maximum of 8. Image assets are also contained in Chroma Hub, so before we can render an image we request it from Chroma Hub. Chroma Hub first send 4 bytes with the length of the image, followed by the image as a raw PNG file. Then we store this data for later use, as well as decoding the png to extract the pixel data using libpng. In later render calls, if the image id matches the currently stored image id, we reuse this file instead of requesting the image again from Chroma Hub. After receiving the graphics request and any image assets, Chroma Engine computes the keyframes for the template. This includes the absolute position calculations. The keyframe process is described in the Chroma Artist section. For smooth animations, we use a bezier curve to control the animation timing. Finally each geometry in the page is rendered to the screen using OpenGL. <img src="/assets/chroma-graphics/chroma-engine.png" alt="Chroma Engine">

Chroma Hub

The purpose of Chroma Hub is to synchronize the graphic templates used by Chroma Viz and Chroma Engine instances. Chroma Hub also stores any assets needed by the templates such as images, currently only in png format. Chroma Hub wraps a SQL database, which currently needs to be setup with the schema in hub/chroma_hub.sql . Internally we use the encoding/json Golang package for simplicity to write and read a json format of the database to a file. Chroma Hub implements a REST API for updating/retrieving assets. Chroma Artist is currently the only application which makes POST requests, to import/update templates and assets. Chroma Artist/Viz and Engine using GET requests to retrieve assets.

Chroma Artist

Chroma Artist provides a UI for designing templates which can be imported to Chroma Hub and used by Chroma Viz. The key difference between Chroma Viz and Chroma Artist is Chroma Artist is used to manipulate the geometry hierarchy of a template, and create Keyframes for the template. In the discussion of Chroma Engine, we omitted the discussion of relative coordinates. To enable easier manipulation of graphics, each geometry has a parent geometry. This gives the geometries a tree structure, and position of a geometry is relative to the position of the parent geometry. Chroma Artist gives an easy interface to specify this tree structure, which Chroma Engine rebuilds to calculate the absolute positions. An example of this functionality is a simple lower frame super, which contains a rectangle for the background, two text geometries parented to the rectangle, and a circle as a logo placeholder also parented to the rectangle. The designer could set the default position, width and height of the background rectangle, aswell as the position of the text. Since the text geometries and circle are parented to the rectangle, to move the graphic we only need to change the rectangle position. In Chroma Artist, the following image shows an example of this graphic.
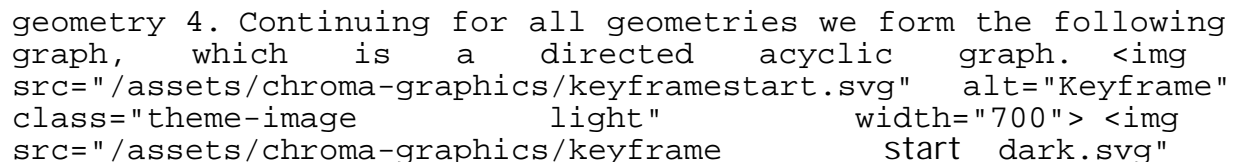![keyframe](/assets/chroma-graphics/keyframeartist.png)
Currently the rectangle is static and the width of the rectangle needs to be updated when the text changes. Keyframing allows us to animate the graphic and have the width of the rectangle be linked to the text width. The keyframing system is based on a directed acyclic graph. We begin with the simplest case of no keyframes which is used to calculate the absolute positions from relative position. This will be extended to construct keyframes. Consider the following table which represents some attributes of the graphic we described above, <img src="/assets/chroma-graphics/keyframeimg.svg" alt="Keyframe" class="theme-image light" width="700"><img src="/assets/chroma-graphics/keyframe img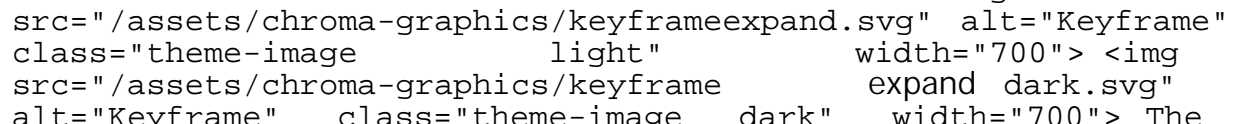 dark.svg" alt="Keyframe" class="theme-image dark" width="700"> The x position of geometry 4 depends on the x position of geometry 1 (it's parent) and its own relative x position. We indicate this dependency with an arrow from the x position of geometry 4 to the x position of geometry 1 and the relative x position of

geometry 4. Continuing for all geometries we form the following graph, which is a directed acyclic graph. <img src="/assets/chroma-graphics/keyframestart.svg" alt="Keyframe" class="theme-image light" width="700"> <img src="/assets/chroma-graphics/keyframe start dark.svg" alt="Keyframe" class="theme-image dark" width="700"> At each node of this graph, we want to compute the value of the node by combining the values of the dependencies in some way. To do this we store a function at each node, in the case of relative positions this function evaluates the sum of the value for each child node of the current node. But we need to evaluate each node only once each child node has been evaluated. This problem is known as topological sorting. This excellent video by William Fiset describes topological sorting of directed acyclic graphs ([Topological Sort](https://youtu.be/eL-KzMXSXXI?si=mG9) Tj ET BT We can make the rectangle width dynamic by adding the following dependencies, and adding a function to the rectangle node which evaluates the maximum of the value of each child node. <img src="/assets/chroma-graphics/keyframeexpand.svg" alt="Keyframe" class="theme-image light" width="700"> <img src="/assets/chroma-graphics/keyframe expand dark.svg" alt="Keyframe" class="theme-image dark" width="700"> The keyframing process in Chroma Artist consists of two objects, Frames and Keyframes. Frames have an index 1, 2, ... up to some fixed number $ n $, and contain a collection of keyframes. For each frame, we create the table of attributes for each geometry show above. A Keyframe is a way to add dependencies and evaluation functions to the graph. When animated, geometry attributes are interpolated linearly between the frames. Current types of keyframes are

Set Frame: Set the value of an attribute of a geometry in a specific keyframe. In the graph this updates the value of a node and doesn't add any dependencies.

User Frame: Similar to set frame, except the value from the template or page when the graphic is animated on is used. This adds an edge which points to the attribute values set by the user, and is the default for non keyframed attributes.

Bind Frame: Use a value computed in a keyframe. This adds an edge to another node in the graph, and the current node simply takes the value of the single child node.

 Additionally we can set a keyframe to be an expand keyframe, currently only supported for User Frame rectangles and attributes width or height. This is what we used above, we add an edge to the upper x position and upper y position for width and height respectively for each child geometry, and evaluates the maximum of all child nodes. The only restrictions on keyframes is they cannot create cycles in the graph. Doing so makes evaluating them ambiguous, so Chroma Engine terminates if it receives a template with a cycle in the keyframe graph. Putting these together we can create the following graphic. <video width="720" controls>

```
    <source src="https://github.com/jchilds0/chroma-viz/raw/main/data/artist

</video>              <link                      rel="stylesheet"
href="/assets/chroma-graphics/style.css">
```