Concepts and API

The following static resources exist for the lifetime of the renderer, and have their state affected by draw calls:

- color texture
- depth texture
- clip min texture
- clip max texture
- uniform buffer (note: may not exist if device supports push constants)
- glyph vertex buffer (note: may not exist if device supports push constants)
- glyph cache texture

The following dynamic resources can be created/loaded and unloaded dynamically:

- image
- image array
- mesh
- font (note: unloading fonts may not be supported)
- layed out text block

The following draw modifiers exist, and may be different types for 2d vs 3d (n-dimensional):

- transform: apply an affine transformation to draw objects before drawing them
  - represented as a (n+1)x(n+1) homogenous transformation matrix newtype, with constructors for: translate by a vector, scale by a vector, rotate by a quaternion
- color: multiply the color of draw objects by the given color, including an alpha component, before drawing them
  - components represented by u8
- clip: discard any parts of a draw object that lie in the given half-plane or half-volume before drawing them
  - represented as an (n+1)-dimensional vector newtype, wherein fragments are discarded if the inner product of their homogenous coordinate with the given vector is negative
  - ^-- that newtype has constructors representing (min|max)_(x|y|z…)
  - it is invalid for a clip vector to be entirely zero

Draw modifiers form a stack, wherein objects drawn are considered to be modified by each modifier in the stack from the top of the stack down to the bottom before being stamped onto the screen. In practice that's not quite what happens, but what does happen produces an equivalent result.
^-- wait hmm, is that correct? I feel like it's not

The following draw object types exist:

- rectangle: a solid white rectangle in the XY plane, extending from the origin to <1,1> (although it may be different after the application of modifiers)
- image: a pre-loaded image resource drawn as a rectangle in the XY plane, extending from the origin to <1, 1> (although that may be different after the application of modifiers), with image UV coordinate start and extent being incorporated into the draw call, and with "repeat" behavior for if UV coordinates extend the [0, 1] range.
- text: a pre-computed layed out text block resource drawn as white rasterized pixels with a 1:1 relation between canvas units and pixel units

- mesh (only for 3d): a pre-created mesh resource, containing per-vertex and per-triangle data, (TODO which direction?), used in combination with an pre-loaded image array to sample from, with image array index being a per-triangle property but UV coordinates within the image layer, as well as things like position and color, being a per-vertex property

A draw program is a sequence of draw instructions, with certain validity constraints, and entirely describes how to draw a single frame, assuming the context of any dynamic resources referenced by the draw program having already been loaded and not unloaded.

The following draw instructions exist:
- Push a modifier to the modifier stack (2d and 3d versions)
- Draw some object (2d and 3d versions)
- Begin 3d drawing

There is no draw instruction for popping a modifier from the modifier stack. Instead, each draw instruction carries with it the length of the modifier stack immediately before the instruction takes effect. As such, if an instruction carries a modifier count less than the current size of the actual modifier stack, it should be considered to implicitly be preceded by a number of modifier stack pops equal to the difference. It is not valid for an instruction to carry a modifier count greater than the current size of the actual modifier stack.

The base of the stack is considered to be in "2d mode," wherein only 2d modifiers and draw objects are valid to use. These are ultimately drawn to a 2d canvas, wherein each unit is a physical pixel, the origin is the top-left corner of the visible area, the X axis extends rightwards in the visible area, and the Y axis extends downwards in the visible area. The extent of the visible area is set as part of constructing the renderer, and can be changed afterwards.

The "begin 3d drawing" instruction essentially bridges the stack from 2d mode to 3d mode. The instruction includes a view-projection matrix to convert from 3D space to 2D+depth space, which combines with modifiers above and below it in the stack. The "begin 3d drawing" instruction is essentially considered to be itself a modifier in the stack, as far as indexing goes, and as such, a 3d drawing sequence ends through the usual implicit popping mechanism. The vp matrix (view-perspective) is a newtype around a 4x4 homogenous transformation matrix, with constructors for perspective and ortographic projections. Some way to bridge from 3d back to 2d may be added in the future.

When in 3d mode, only 3d modifiers and draw objects are valid to use. It's worth noting that all 2d draw objects are also 3d draw objects, which simply begin embedded into the XY plane until and unless transformed out of it, but not all 3d draw objects are also 2d draw objects, such as a mesh.

TODO: origin and axis direction of UV coordinates and how they related to 2D and 3D?

In 2D mode, draw calls are essentially stamped over whatever is previously drawn (with the exception of the canvas being wiped every frame), and blended together in the case of full or partial transparency. In 3D mode, draw calls are combined using a depth texture, so that the closest fragment to the screen is visible. If parts of the 3D scene are left fully or semitransparent, the entire 3D scene is stamped over what is previously drawn in 2D as if the entire projected 3D scene is a 2D object. Later 3D draw calls will blend with earlier 3D draw calls if fragments are not discarded for depth reasons, but nevertheless, the user must ensure that semitransparent 3D objects are drawn after other objects in the same 3D scene which are visually behind them from the perspective of the camera, or they will not blend correctly.

Implementation

All draw calls work roughly along these lines:

- Initial vertices are produced (from vertex buffers, uniforms, hard-coded values into shaders, push constants…)
- Vertex homogenous positions are multiplied by a cumulative transformation matrix
- Rasterization occurs
- Each on-screen fragment's Z value is compared with the sample of the clip min and clip max textures at the same XY coordinates. If fragment Z is less than clip min or greater than clip max, the fragment is discarded. As we will see, this is used to implement clipping. These textures may hold positive or negative infinity values, and the system is expected to work correctly in such cases.
- Depth texture testing and writing may or may not be enabled for a particular draw call.
- The fragment color is multiplied by the cumulative color modifier before being written.

Under the hood, all transforms and clips are converted to 3d, even if they were initially 2d. Also, all draw objects produce (homogenous) 3d coordinates, even if the were initially 2d.

The cumulative transform matrix at any point in the program is calculated by simply multiplying all active transform modifier matrices together. The cumulative color modifier at any point in the program is calculated by simply multiplying all active color modifiers together.

Each individual clip vector is multiplied by the *transpose* of all transform matrices underneath it in the modifier stack to produce its corresponding post-transform clip vector, so that clipping can be correctly performed always after vertex transformation. However, it's fundamentally not possible to simply combine multiple clip vectors into one.

Rather, before each draw call, unless the following work can be reused due to the active post-transform clip vector set being unchanged since this was last done, the following is done:

- clip min is cleared to negative infinity
- clip max is cleared to positive infinity

- for each active clip vector, a shader is run over every fragment in either the clip min or clip max texture:
    - let the post-transform clip vector be <a, b, c, d>
    - if c > 0, set every element of clip min to the max of the current value and -(ax+by+d)/c
    - if c < 0, set every element of clip max to the min of the current value and -(ax+by+d)/c
    - if c == 0, either will actually work, and it will involve infinities

When drawing in 3d mode, a depth texture is tested against and written to. When drawing in 2d mode, this is not the case. Upon a “begin 3d drawing” instruction being pushed to the stack, the depth texture is cleared to 1.

GUI Area System

k