The CUGL Scene Graph

Introduction

CUGL is designed to be a data-driven game engine. All assets and other information needed to display a scene to an end-user can be defined and should be defined in a JSON manifest (by default a file named assets.json). This allows for the work of scene and level design to be encapsulated and developed separately and allows for easy asset swapping. More importantly, it allows for all of this without requiring anyone edit the actual codebase of a project or sit through the build times associated with editing said code. If everything has been done correctly, the only filename included in the code itself should be that of assets.json or other manifests that can lead CUGL to everything else.

Aside from using this data-driven manifest to load fonts, images, and sounds, CUGL also uses it to define the scene graph. A Scene Graph is a common data structure in many game engines and graphics applications and serves as a hierarchical collection of “nodes” that defines the display and (some) behaviour of the scene. The key is that each of these nodes can be effectively treated as the root of a new scene graph. It can have child nodes that can have child nodes and so on. From a UI perspective, these nodes can be classic widgets such as sprites, buttons, sliders, progress bars, etc.. They can also be project-specific custom nodes that represent an encapsulation of some project specific functionality and displayed information.

The 2D scene graph of CUGL also comes with multiple forms of automated layout managers that allow for dynamic placement and resizing of nodes to account for the various screen sizes and aspect ratios that come with cross-platform application development.

Most importantly, because all of CUGL is exposed, it is both feasible and encouraged to add additional asset loaders, node types, layout managers, and the like to suit the needs of individual projects. The system is designed with extensibility in mind and if a desired piece of functionality is not outlined within this document, the last sections provide guidance for adding such new features in ways that can continue to leverage the scene graph and other data-driven tools that CUGL provides.

The Many Pieces of assets.json

Each of the following represents a different top-level JSON object in assets.json and explains what it holds. Many of these are optional and they do not have to be the only top-level manifests included, but those provided here are almost always used in full-scale projects.

Fonts

All fonts are loaded into the AssetManager through a FontLoader. This loader allows for allocating fonts from the associated TrueType (.ttf) files. At load time, the TrueType file and the font size must both be specified. Because of this, it may be desirable to load a font asset multiple times at different sizes. When doing this, it may be beneficial to adjust the character set, as the size of the font atlas texture is determined by both the font size and the character set.

When loading a font in JSON, it is defined via a JSON object with the following fields:

- “file” (string)
    - “file” specifies the relative pathname to the desired font. This includes the file’s full name and extension.
- “size” (int)
    - “size” specifies the size for this font.
- “charset” (string)
    - “charset” is a string containing every character to be included when loading the font. If this is not included or is set to the empty string, the default character set containing all of the ASCII characters will be used.

Multiple of these objects may be listed within the top-level “fonts” object in the JSON. Each of them should be named uniquely. This will be the key by which the fonts are accessed from the asset manager.

Jsons

The AssetManager can load JsonValue objects through a JsonLoader (effectively a wrapper to JsonReader). This allows for uploading arbitrary JSON data and fetching it later from the AssetManager.

When loading another JSON file in the scene graph JSON it is not defined in an object but rather as a field within the top-level “jsons” object. Each field should be named uniquely as this is how the JsonValue will be looked up from the AssetManager later. The value for this field is the relative pathname to the .json file that should be loaded.

The JsonLoader is a good place to start when loading project specific data into CUGL and serves as the best jumping-off point when adding custom loaders for this data.

Textures

All textures are loaded into the AssetManager by a TextureLoader. This loader allows for loading any image file that it supported by SDL by default. A texture asset is identified by both its source file and its texture parameters. Because of this, it may be necessary to load a texture asset multiple times, though this is potentially wasteful of memory. Changing the parameters for a texture asset will change the asset parameters in the loader as well.

When loading a texture in JSON, it is defined via a JSON object with the following fields:

- “file” (string)
    - “file” specifies the relative pathname to the desired texture. This includes the file’s full name and extension.
- “mipmaps” (boolean)
    - “mipmaps” specifies whether to generate mipmaps for this texture. Mipmaps allow for better texture filtering when resizing a texture below its actual size but increase the size overhead of a texture in memory.
- “minfilter” (string) one of ‘nearest’ | ‘linear’ without mipmaps
  or one of one of ‘nearest-nearest’ | ‘linear-nearest’ | ‘nearest-liner’
  | ‘linear-linear’ with mipmaps
    - “minfilter” provides the minification filter used by OpenGL when drawing the texture smaller than it is loaded. The options for this filter change depending on whether mipmaps have been generated. Without mipmaps, the options are ‘nearest’ or ‘linear’.
      Nearest uses the value of the texel that is nearest (in Manhattan distance) to the center of the pixel being textured.
      Linear uses the weighted average of the four nearest texels to the center of the pixel being textured.
      With mipmaps, the options are ‘nearest-nearest’, ‘linear-nearest’, ‘nearest-linear’, and ‘linear-linear’.
      Nearest-Nearest chooses the mipmap that most closely matches the size of the pixel being textured and uses the Nearest rule to produce a texture value.
      Linear-Nearest chooses the mipmap that most closely matches the size of the pixel being textured and uses the Linear rule to produce a texture value.
      Nearest-Linear chooses the two mipmaps that most closely match the size of the pixel being textured and uses the Nearest rule to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.
      Linear-Linear chooses the two mipmaps that most closely match the size of the pixel being textured and uses the Linear rule to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

(The first half determines the rule within a mipmap, the second half determines which mipmaps to use).

- “magfilter”: (string) one of ‘nearest’ | ‘linear’
  - “magfilter” provides the magnification filter used by OpenGL when drawing the texture larger than it is loaded. It may either be ‘nearest’ or ‘linear’.
    Nearest uses the value of the texel that is nearest (in Manhattan distance) to the center of the pixel being textured.
    Linear uses a weighted average of the four texels that are closest to the center of the pixel being textured.
- “wrapS”: (string) one of ‘clamp’ | ‘repeat’ | ‘mirrored’
  - “wrapS” provides the s-coord wrap rule. This defines the behavior when sampling from the texture outside of its bounds horizontally.
    Clamp causes the texture to use the pixel value at the outer edge of the texture; use this when importing tiles from a tileset to avoid aliasing lines.
    Repeat causes the texture to sample from the other side of the texture in a modular way; use this when you want to tile the same image over a larger area continuously.
    Mirrored functions identically to Repeat except that every other time the texture wraps it will be drawn backwards.
- “wrapT”: (string) one of ‘clamp’ | ‘repeat’ | ‘mirrored’
  - “wrapT” provides the t-coord wrap rule. This defines the behavior when sampling from the texture outside of its bounds vertically.
    Clamp causes the texture to use the pixel value at the outer edge of the texture; use this when importing tiles from a tileset to avoid aliasing lines.
    Repeat causes the texture to sample from the other side of the texture in a modular way; use this when you want to tile the same image over a larger area continuously.
    Mirrored functions identically to Repeat except that every other time the texture wraps it will be drawn backwards.

See https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/glTexParameter.xml for documentation on how these parameters translate to those used by OpenGL.

Sounds

All sound assets are loaded into the AssetManager through a SoundLoader. A full explanation of which audio files are supported may be found here, but as a general rule, sound assets should be WAV files as there is not benefit for compression. Sounds are loaded both with a pathname to the source and a default volume at which to play the sound.

When loading a sound in JSON, it is defined via a JSON object with the following fields:

- "file" (string)
  - "file" specifies the relative pathname to the desired sound. This includes the file's full name and extension.
- "volume" (float)
  - "volume" specifies the default volume for this sound when played.

Scenes

The actual scene graph is loaded into CUGL through the hierarchical objects that live within the top-level JSON object “scenes.” These scenes are loaded in a preorder fashion through the SceneLoader. As UI widgets typically require fonts, images, or other data to be loaded already, scenes should always be the last elements loaded in a loading phase.

When loading a node from JSON, it is defined as a JSON object with the following fields:

- “type” (string)
    - “type” specifies the node type (a Node or any subclass thereof). These nodes are documented thoroughly in the next section, but a list of values for “type” is provided here with a brief overview.

“None”	The base Node type
“Image”	An image (PolygonNode) type
“Poly”	A (complex) PolygonNode type
“Path”	A PathNode type
“Wire”	A WireNode type
“Animate”	An animation node type
“Nine”	A nine-patch type
“Label”	A Text label (uneditable) type
“Button”	A button type
“Progress”	A progress bar type
“Slider”	A slider type
“Textfield”	A single-line text field type

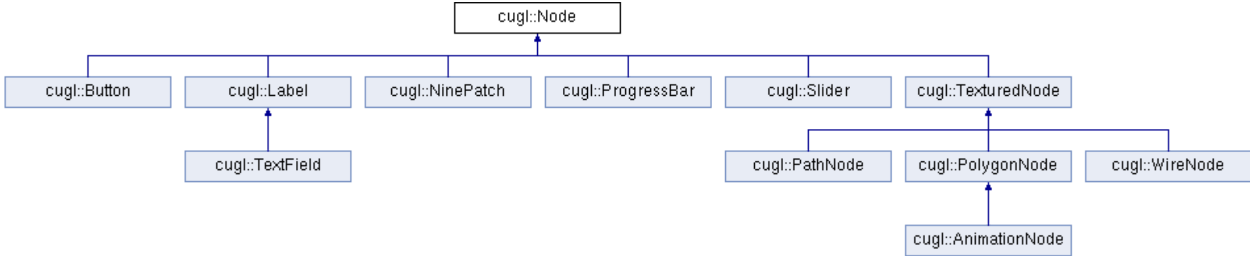
    If a custom node type has been added, then a value to indicate that type could also be added here.

- “data” (json)
    - “data” is a json object that provides all additional data (images, labels, etc.) that define the widget. This object has a format specific to the node type defined in “type”.

- “format” (json)
    - “format” is a json object that defines the layout manager to use for this node. This layout manager will apply to all of this node’s children. If left out, the default layout manager using absolute position is assumed. Additional fields within “format” beyond its internal “type” are all layout manager dependent.
- “layout” (json)
    - “layout” is a json object that specifies how a node should be placed within its parent’s layout manager. This layout change is applied after parsing “data” and will override any settings there. The fields of “layout” are dependent on the type of layout manager specified by its parent.
- “children” (json)
    - “children” is a json object that defines additional nodes for each child of this node. Each child is provided as its own named attribute within “children”. The nodes defined here will be loaded hierarchically along with all of their children and then added as children to the node defined here. This is where the “graph” part of the scene graph comes from.

The next two sections “Nodes Elaborated by Type” and “Using Layouts” describe the type-specific attributes of the “data”, “format”, and “layout” blocks.

Nodes Elaborated By Type



Node

Overview

From the CUGL 1.2 documentation:

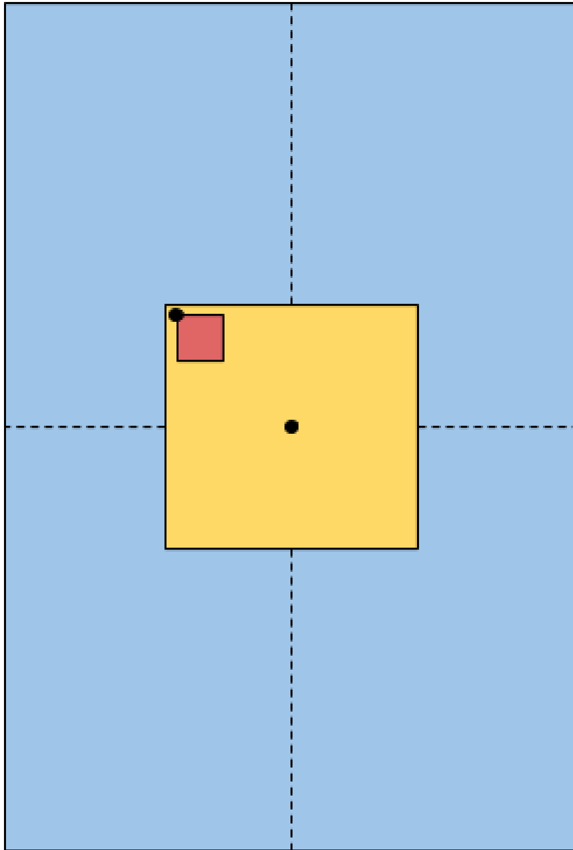
> A base node is a rectangular space that can contain other (transformed) nodes. Each Node forms a its own coordinate space. All rendering takes place inside of this coordinate space. By default, a Node does not render anything, but it does provide rendering support for subclasses via the draw() method.

What this means in larger detail:

> Nodes form what those familiar with computer graphics or the inner workings of other game engines would consider the 2d analog to a transform hierarchy. Nodes have children, those nodes have children, etc., and each node within this hierarchy can apply its transformation to all children recursively such that when we work with a single node and its children, we can abstract away everything higher in the hierarchy as the current node and instead treat it as a transformed root of our tree. More simply put, nodes give us a way to arrange 2d regions of the scene with regard to specific parent objects. A node's coordinate space is determined through a combination of its position and its anchor. The node's anchor, represented as x and y floating point values between 0 and 1[1], defines where this node's anchor position belongs proportionate to its space. 0 represents the far left or bottom edge of the node and 1 represents the far right or top edge of the node. This anchor is then lined up with the parent's coordinate space such that it is at the x,y coordinate defined by "position" in its parent's space. Note that the origin for a node is always in its lower left corner, no matter where the anchor is. All scaling and rotating is done around the anchor point.

---

[1] There is nothing preventing anchors higher than [1,1] or lower than [0,0], but these anchors will be outside the bounding box of the node.



Blue Box
Parent: None
(root of what is shown here)

Yellow Box
Parent: Blue Box
Anchor: [.5,.5]
Position: $(w_b / 2, h_b / 2)$

Red Box:
Parent: Yellow Box
Anchor: [0, 1]
Position: $(5, h_y - 5)$

Figure 1: Here is an example of node anchors and positions in use. The values w_b and h_b are the width and height of the blue box respectively while h_y represents the height of the yellow box. This example shows how the red box can be positioned with exact padding relative to the upper left corner of the yellow box while the yellow box is positioned perfectly centered relative to its parent, the blue box. Notice that we do not have to go through the complex process of determining where the origin of the red box resides in the coordinates of the blue box or screen space in order to draw it correctly. The anchors have been displayed here as black points for the convenience of the reader.

Initializing from JSON with data

The “data” block of any Node accepts the following fields:

- “position”: (two-element number array) [x, y]
  - “position” provides the coordinate location of this node’s anchor within the space of its parent. It is x units left and y units up.
- “size”: (two-element number array) [x, y]
  - “size” defines the dimensions of the node. This affects how the anchor points of children resolve.
- “anchor”: (two-element number array representing an anchor point) [x, y]
  (0 to 1 for inside bounds)
  - “anchor” defines where this node’s anchor is placed within its bounds. This is the point that will be lined up with the coordinates defined by position. Anchors are represented similarly to texture coordinates in that they represent a fractional distance across the node rather than a pixel-coordinate location. [0,0] os the bottom-left, [1,1] is the top-right, [0.5, 0.5] is the horizontal and vertical center, and so on. Anchors with values outside of the [0,1] range can be used to align nodes based on a point outside of its bounds.
- “scale”: (either a two-element number array) [sx, sy]
  OR (a single number) s
  - “scale” defines the amount to scale the node (and all of its children) in the hierarchy. This can be a uniform scale in both directions or be a different scale for the two dimensions. Nodes are scaled outwards from the anchor.
- “angle”: (a number in *degrees*) q
  - “angle” defines the number of degrees counter-clockwise to rotate this node about its anchor.
- “visible” (boolean) true | false
  - “Visible” determines whether or not this node and its children are drawn when traversing the hierarchy.

The scene from Figure 1 may have been defined by a JSON similar to the following:

```
"blue_box": {
  "type": "Node",
  "data": {
    "size": [200, 300],
    "visible"” true
  }
  "children": {
    "yellow_box" : {
      "type": "Node",
      "data": {
        "anchor": [0.5, 0.5],
        "position": [100, 150],
        "size": [80, 80]
        "visible": true,
      },
      "children": {
        "red_box": {
          "type": "Node",
          "data": {
            "anchor": [0, 1],
             "position": [5, 75],
            "size": [10, 10]
            "visible": true,
          }
        }
      }
    }
  }
}
```

Note that the base node type is not drawn to the screen, so in order to draw these boxes to the screen with the correct color we would need to use another kind of node, such as the TexturedNode. This snippet is only meant to serve as an example on how the coordinate systems can be used in the hierarchy.

Also notice that not every attribute is used in every data block. Most fields that can be included in the “data” block of a Node are not required.

Button

Overview

The Button node type represents a simple clickable button. It is either two nodes (one for up, and one for down) that swap whenever the button is pressed, or it is a single node that changes color on press. If a button tracks its own state (see the full CUGL documentation for a description of how), it is classified as either a normal button or a toggle button. A normal button is down only while it is pressed; a toggle button changes state when it is pressed and retains that state until it is pressed again. THe clickable region of a button can be defined as any arbitrary polygon, allowing click response to match more complex images.

Initializing from JSON with data

In addition to all fields from Node, the "data" block for Button also takes the following fields:

- "**up**": (a JSON object defining a scene graph node)
    - "up" defines the scene graph node that is rendered when the button is in its up state. This attribute is required.
- "down": (a JSON object defining a scene graph node)
  OR (a 4-element integer array with values from 0..255) [r, g, b, a]
    - "down" defines the scene graph node that is rendered when the button is in its down state. Alternatively, it can be a color used to tint the up node to represent that the button is down.
- "pushable": (An even array of polygon vertices[2])
    - "pushable" defines the polygonal region in which the button can be pushed. These vertices will be converted into a polygon using SimpleTriangulator.

[2] Polygon vertices are represented in the JSON as numbers.

Label

Overview

A Label is a node representing a single line of text. By default, the content size of a label is set to be just large enough to render the text that is given and will automatically resize if the text it changed. If the size is set to be larger than what the text needs, it will be positioned within the label based on the horizontal/vertical alignments and the text bounds. If the size is reset to be smaller than what the text needs, it will also place it according to the alignment rules, but the text may be cut off in rendering.

If the background color is not clear, then the label will have a colored backing rectangle extending from the origin to the content size in the label's node space.

To display the text, you need a Font. The rendering style of the font will depend on whether or not it has an atlas. For performance reasons, it is highly recommended to use a font atlas unless the project calls for a very small number of labels that never have their text changed at all.

Initializing from JSON with data

In addition to all fields from Node, the “data” block for Label also takes the following fields:

- “**font**”: (string)
  - “font” provides the name of a previously loaded font asset to be used by this label. This attribute is required.
- “text”: (string)
  - “text” provides the initial label text.
- “foreground”: (4-element integer array) [r, g, b, a]
  - “foreground” provides the color used to render the text itself.
- “background”: (4-element integer array) [r, g, b, a]
  - “background” provides the color used to fill the node’s space behind the text.
- “padding”: (A two-element float array) [x, y]
  - “padding” provides the amounts of horizontal and vertical padding by which to move the text away from the alignment edge. For example, if the halign is set to 'left' then the x value is the amount of space that the label text is padded to the right and if the halign is set to 'right’ then the x value is the amount of space that the label text is padded to the left. The x value is ignored for 'center’ or 'hard center’. The y value is the vertical equivalent of the x value and ignored for 'middle’ or 'true middle’.
- “halign”: (string) one of ‘left’ | ‘center’ | ‘right’ | ‘hard left’ | ‘true center’ | ‘hard right’
  - “halign” provides the horizontal alignment of the text. This effectively sets the horizontal anchor of the text relative to the label. 'hard left’, 'true center’, and 'hard right’ ignore spaces on the outside of the text that would otherwise influence alignment.
- “valign”: (string) one of ‘top’ | ‘middle’ | ‘bottom’ | ‘hard top’ | ‘true middle’ | ‘hard bottom’
  - “valign” provides the vertical alignment of the text. This effectively sets the vertical anchor of the text relative to the label. 'hard bottom’ ignores the font descent and puts the lowest hanging character edge at the bottom of the label. 'hard top’ ignores the font ascent and puts the highest reaching character edge at the top of the label. 'true middle’ ignores the font height and instead centers text relative to the highest and lowest reaching character edges.

TextField

Overview

The TextField is a subclass of Label and represents a single line text field. When a text field is "focused" (selected to receive keyboard events), the application user will be permitted to edit the text within the text field. When focused, it displays a blinking cursor with the current edit position. Like a Button, a TextField can track its own focus state via the activate(Uint32) method.

Initializing from JSON with data

TextFields are initialized from the scene graph JSON in the exact same way as Labels. They have no additional fields.

NinePatch

Overview

The NinePatch node type displays a ninepatch (sometimes also referred to as a nineslice) image. Ninepatch images are commonly used for resizable UI elements because they preserve the aspect of the four corners while allowing the central regions to stretch. In practice the resized portion of the ninepatch texture is left monochromatic along the stretched axis to minimize the visual effects of stretching these regions.

If the rectangle used for the interior is left undefined, it assumes a degenerate ninepatch with a one-pixel middle at the center of the image.

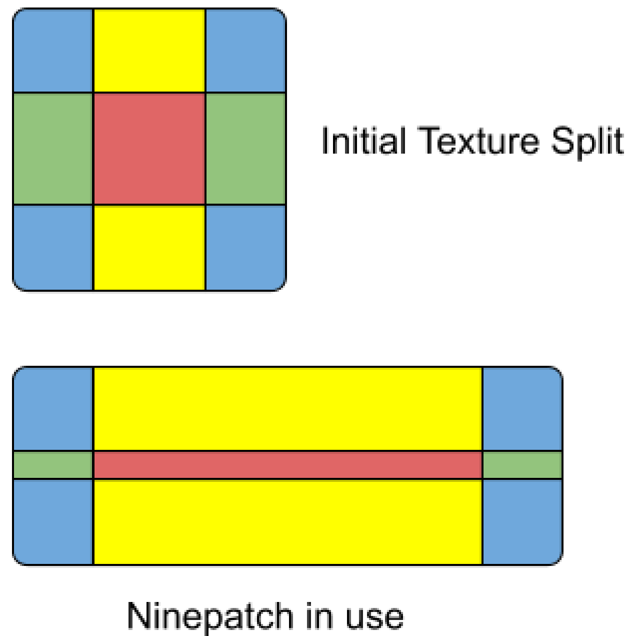


Figure 2: Here is an example of a NinePatch in use. Notice that the four corners (blue) are not stretched at all, the central top and bottom regions (yellow) are only stretched horizontally, the middle left and right regions (green) have been squashed vertically, and the central region (red) has been resized along both axes.

Initializing from JSON with data

In addition to all fields from Node, the “data” block for NinePatch also takes the following fields:

- “texture” (string)
    - “texture” provides the name of a previously loaded texture asset to be used by this NinePatch.
- “interior” (4-element array of numbers) [x, y, width, height]
    - “interior” provides the bounds of the interior of the NinePatch as specified by pixel coordinates. (x, y) is the lower-left corner of the interior region with $(x + width,\ y + height)$ being the upper-right corner.

ProgressBar

Overview

The ProgressBar node represents an (animating) progress bar. This progress bar may either be represented via a texture or by a simple colored rectangle. If it is a texture, the foreground texture will be sampled left to right and the maximum horizontal texture coordinate will be the percentage of the progress bar. So if the progress bar is at 50%, the progress bar will draw the left half of the pixels in the foreground texture, filling the left half of the foreground region. Additionally, when using a texture, it is possible to specify end-cap textures, allowing for progress bars that are not purely rectangular.

Initializing from JSON with data

In addition to all fields from Node, the “data” block for ProgressBar also takes the following fields:

- “foreground” (string)
    - “foreground” provides the name of a previously loaded texture asset to be used for the progress bar mid-section.
- “background” (string)
    - “background” provides the name of a previously loaded texture asset to be used as the background for the progress bar.
- “left_cap” (string)
    - “left_cap” provides the name of a previously loaded texture asset to be used as the left cap of the filled portion of the progress bar.
- “right_cap” (string)
    - “right_cap” provides the name of a previously loaded texture asset to be used as the right cap of the filled portion of the progress bar.

Slider

Overview

The Slider node represents a slider. It allows the user to drag a knob to select a value.
The slider itself is defined by this "knob" (a Button node) and a path (a Path node). If these are not specified, the class constructs a simple slider with a circle on a line.
The most important attribute for a slider is the bounds attribute; this rectangle defines the slidable region inside the Path node. This allows for complex path nodes with tick marks and other features that would prevent the slider from being centered in the node. It also allows sliders to be defined with any orientation. The bottom left corner is the minimum value with the top right as the maximum.
A tick interval can also be set to snap the slider to predefined values. The class will not automatically display tick marks. To display tick marks, add them to the path node.
Like Button, the Slider can track its own state via the activate(Uint32) method.

Initializing from JSON with data

In addition to all fields from Node, the "data" block for Slider also takes the following fields:

- "bounds": (A 4-element array of numbers) [x, y, width, height]
  - "bounds" defines the rectangular bounds of the slidable element. This is the region within which the knob can be slid.
- "range": (A 2-element array of numbers) [min, max]
  - "range" defines the minimum and maximum values for this slider.
- "value": (a number)
  - "value" defines the initial value of the slider.
- "tick" (a number > 0)
  - "tick" provides the tick period of the slider. The slider will only snap to these ticks if the "snap" attribute has also been set.
- "snap" (boolean) true | false
  - "snap" determines whether or not to snap to the nearest tick.
- "knob" (JSON object)

    - “Knob” provides the JSON definition of the Button object that is used for the slidable element within the slider. It will be allowed to move along the path while in its down state.

- “path” (JSON object)
    - “path” provides the JSON definition of the PathNode object that is used for the slider track. “knob” is moved along “path.”

TexturedNode

Overview

TexturedNode is an abstract scene graph node representing any textured shape. Rather than instantiate one directly, any TexturedNode must instead be instantiated as a PolygonNode, PathNode, or WireNode (or any subclass of one of these). Some things are the same for all textured nodes:

1) The node shape is stored as a polygon. Polygons are specified in image coordinates (origin at the bottom left, each pixel is one unit). A polygon with vertices (0,0), (width,0), (width,height), and (0,height) would be identical to a sprite node; however, a polygon with vertices (0,0), (2*width,0), (2*width,2*height), and (0,2*height) would tile the sprite (given the wrap settings) twice, both horizontally and vertically.
2) The content size of a textured node is defined by the size (but not the offset) of the bounding box. The anchor point is relative to this content size. The default anchor point in TexturedNode is (0.5, 0.5). This means that a uniform translation of the polygon (in contrast to the node itself) will not move the shape on the screen, but rather change the part of the texture it uses.

You can disable either of these features in code by setting the attribute “absolute” to true. This will place polygon vertices in their absolute positions in node space and ignore the anchor, treating it instead as [0,0]. This allows for drawing vertices directly into the coordinate space.

Initializing from JSON with data

In addition to all fields from Node, the “data” block for TexturedNode also takes the following fields:

- “texture” (string)
  - “texture” provides the name of a previously loaded texture asset to be used by this node.
- “polygon” (an even array of polygon vertices)
  - “polygon” provides an array of vertex coordinate pairs to be used to define the triangulation of this node. It is in the form of $[x_0, y_0, x_1, y_1, x_2, y_2, ...]$.
- “indices” (unsigned int array of length 3n)
  - “indices” defines the triangles of the polygon by indexing vertex pairs from “polygon”. The triangulation [0, 1, 2, 2, 1, 3] defines the two triangles $\{(x_0, y_0),\ (x_1, y_1),\ (x_2, y_2)\},\ \{(x_2, y_2),\ (x_1, y_1),\ (x_3, y_3)\}$.

All of these three attributes are optional, but it is recommended to at the very least either define the texture or the polygon.

Figure 3: This is an example of the same triangulation defined above used to represent a simple quad. Traditionally, we order the indices that define a triangle such that they wind around the outside in a counter-clockwise fashion.

PathNode

Overview

A PathNode represents a path with width.

At first glance, it would appear that this class is unnecessary. A path with width, produced by PathExtruder, is a solid polygon. This polygon can, in turn, be used in conjunction with PolygonNode. However, there are some subtle issues with this. In particular, mitres and joints may mean that a PathNode and WireNode at the same position may not line up with one another. This is undesirable. Hence, this is a special polygon node that takes into account that it is an extruded path.
One of the side effects of this is that the content size of the node is defined by the wireframe path, NOT the extruded path. In the code, getExtrudedContentBounds() provides the bounds of the extruded path (relative to Node space). Additionally, the anchor point is relative to the content size not the extruded size. This means that the extruded path may be to the left of the origin even when the anchor is (0,0).

Because paths have width, it is natural to texture them; however, generally it is recommended to create a path with the degenerate texture (to draw a solid, colored path). Because of this, none of the static constructors for PathNode take a texture, but it is possible to update the texture after node creation.

Initializing from JSON with data

In addition to all fields from both Node and TexturedNode, the “data” block for PathNode also takes the following fields:

- “stroke”: (number)
    - “stroke” specifies the stroke width of the path.
- “joint” (string) one of ‘mitre’ | ‘bevel’ | ‘round’
    - “joint” defines the style of joint used: one of either ‘mitre’, ‘bevel’, or ‘round’. If left off, it defaults to ‘none’. ‘none’ will make the path look like a sequence of links. ‘mitre’ will use a mitre joint, ideal for sharp corners. ‘bevel’ will use a bevel joint, ideal for smoother paths. ‘round’ will use a round joint (best for smoothing out paths with sharp corners).
- “cap” (string) one of ‘square’ | ‘round’
    - “caps” specifies the style of cap used: either ‘square’ or ‘round’. If left off, it defaults to ‘none’. None will lead to a path with no end cap, terminating at the end vertices. ‘square’ is the same as no cap, but pads the end by the stroke width. ‘round’ ends the path with half circles of radius equal to stroke width.
- “closed” (boolean) true | false
    - “closed” specifies whether or not the path is closed. If it is closed, it will loop and therefore not have end caps.

PolygonNode

Overview

A PolygonNode represents a solid 2D polygon textured by a sprite. It is the simplest implementation of TexturedNode and should be the default choice for most nodes that need to display a texture.

Initializing from JSON with data

PolygonNodes are initialized from the scene graph JSON in the exact same way as TexturedNodes. They have no additional fields.

AnimationNode

Overview

An AnimationNode is a special PolygonNode that supports simple filmstrip animation. This class is very similar to PolygonNode, but it treats the texture provided as a sprite sheet and therefore needs a number of rows and columns specified so that it can break up images.
The basic constructors for AnimationNode in the code always set this object as a rectangle the same size as a single frame in the sprite sheet, but it is certainly possible to instead animate the filmstrip over polygons. This can have undesirable effects if the polygon coordinates extend beyond a single animation frame as the basic renderer does not allow wrapping a single frame of a texture atlas (a filmstrip being a special case of a texture atlas). For example, with a filmstrip where each frame has a given width and height, setting the polygon to a triangle with vertices (0,0), (width/2, height), and (width,height) is okay, but the vertices (0,0), (width, 2*height), and (2*width, height) can cause other frames from the filmstrip to also be displayed within the polygon.
It is assumed that frames are ordered left to right along rows with multiple rows ordered from top to bottom.

Initializing from JSON with data

In addition to all fields from both Node and TexturedNode, the “data” block for AnimationNode also takes the following fields:

- “span” (number)
    - “span” indicates the number of frames in the filmstrip. If it is not defined, a degenerate filmstrip of a single frame is assumed.
- “cols” (int)
    - “cols” specifies the number of columns in the filmstrip. If it is not defined, it is assumed that there is only one row (assumption that cols == span).
- “frame” (int)
    - “frame” specifies the index of the starting frame within the filmstrip. The AnimationNode will display the sprite at this frame until the setFrame() method is called with a new frame number.

WireNode

Overview

WireNode represents a wireframe. The wireframes are lines, but they can still be textured. However, generally, it is best to create a wireframe with the degenerate texture (to draw a solid, colored line). Because of this, none of the static constructors in the ode take a texture, but it is possible to update the texture after creation. The node shape is stored as a polygon, and the wireframe shape is determined by the polygon traversal.

Initializing from JSON with data

In addition to all fields from both Node and TexturedNode, the “data” block for WireNode also takes the following fields:

- “traversal” (string) one of either ‘open’ | ‘closed’ | ‘interior’
    - “traversal” indicates the style of traversal used by the PathOutliner.
      If ‘open’, the traversal is in order but does not close the ends.
      If ‘closed’, the traversal is in order and closes the ends.
      If ‘interior’, the traverse will outline the default triangulation.
      The default traversal is ‘closed’.

Using Layouts

One of the worst problems that arises when setting node positions directly is the inability to design interesting scene graphs that look good on devices of different shapes and sizes. For console games that assume a TV and specific aspect ratio, a few different hard-coded options may be sufficient, but for any successful mobile application resizable contents are a must.

For those looking to more dynamically define the size and position of nodes in the scene graph, there are ways to define layout managers in the scene graph JSON that are associated with certain nodes there.

When a layout manager is asked to layout a Node, it searches for those children that are registered with the layout manager. For those children, it repositions and/or resizes them according to the layout information.

Layout information is indexed by key. To look up the layout information of a scene graph node, the name of the node is used as this key. This requires all nodes to have unique names. The SceneLoader prefixes all child names by the parent name, so this is the case in any well-defined JSON file.

There are multiple layout managers that all extend the Layout class. Each provides implementations for the add, remove, and layout methods.

Several layout managers, such as AnchoredLayout and GridLayout make use of anchors; therefore, support has been provided for them in the base class in order to consolidate code.

When specifying layouts in the JSON, two different objects are used. The “format” object is used by the top level parent of a particular layout manager. This “format” block specifies everything necessary to initialize the manager itself. In any child object of this top-level node, the “layout” object is used to add the child to the layout manager after it has been initialized. This “layout” block specifies any special data used by the manager to position the child. The fields of “format” are documented as part of the allocWithData() or initWithData() methods of a layout class, and the fields of “layout” are documented as part of the add() method of this same class.

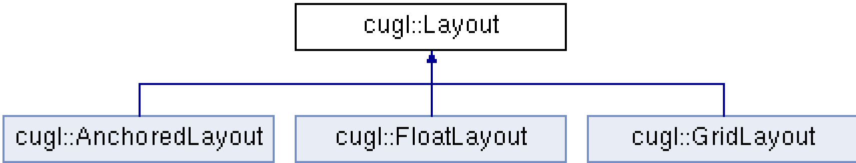
Layout managers may be nested in the same way as nodes. If a node has both a “format” block and a “layout” block, the “layout” block will be used to add it to the parent layout manager while the “format” block will be used to initialize a new layout manager for its children.

The 16 Layout Anchors

Unlike the anchors used to position nodes directly, the anchors used by layouts include independent options for whether to fill horizontally and whether to fill vertically. This gives us 16 anchors instead of 9. It may be beneficial to think of these anchors as tools for stretching child nodes as you would regions of a NinePatch.

Anchor Result by Enumerator Name	
BOTTOM_LEFT	Anchors to the bottom left corner, or position (0,0) in Node coordinate space.
MIDDLE_LEFT	The left side, or position (0,height/2) in Node coordinate space.
TOP_LEFT	The top left corner, or position (0,height) in Node coordinate space.
BOTTOM_CENTER	The bottom side, or position (width/2,0) in Node coordinate space.
CENTER	The middle region, or position (width/2,height/2) in Node coordinate space.
TOP_CENTER	The top side, or position (width/2,height) in Node coordinate space.
BOTTOM_RIGHT	The bottom right corner, or position (width,0) in Node coordinate space.
MIDDLE_RIGHT	The right side, or position (width,height/2) in Node coordinate space.
TOP_RIGHT	The top left corner, or position (width,height) in Node coordinate space.
BOTTOM_FILL	Anchors at y=0, but stretches the width to fill the parent.
MIDDLE_FILL	Anchors at y=height/2, but stretches the width to fill the parent.
TOP_FILL	Anchors at y=height, but stretches the width to fill the parent.
LEFT_FILL	Anchors at x=0, but stretches the height to fill the parent.
CENTER_FILL	Anchors at x=width/2, but stretches the height to fill the parent.
RIGHT_FILL	Anchors at x=width, but stretches the height to fill the parent.
TOTAL_FILL	Stretches the width and height to fill the entire parent.
NONE	No anchor. The layout will not adjust this child.

Layouts Elaborated By Type



FloatLayout

Overview

FloatLayout provides a float layout manager.

Children in a float layout are arranged in order, according to the layout orientation (horizontal or vertical). If there is not enough space in the Node for the children to all be in the same row or column (depending on orientation), then the later children wrap around to a new row or column. New rows are added downwards and new columns are added to the right.

Any children that cannot fit (non-overlapping) into the Node are dropped. Once a child is dropped, no further children will be placed. So an exceptional large child can block the rest of the layout.

Anchor Enumerator for FloatLayout

Anchor Result by Enumerator Name	
BOTTOM_LEFT	In horizontal orientation, this left justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their bottom, and the bottom line will be flush with the bottom of the Node. In vertical orientation, this bottom justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their left, and the left column will be flush with the left of the Node.
MIDDLE_LEFT	In horizontal orientation, this left justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their middle, and the layout will be centered in the Node. In vertical orientation, this centers each of individual columns. In addition, all Nodes in a single column will be aligned by their left, and the left column will be flush with the left of the Node.
TOP_LEFT	In horizontal orientation, this left justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their top, and the top line will be flush with the bottom of the Node. In vertical orientation, this top justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their left, and the left column will be flush with the left of the Node.
BOTTOM_CENTER	In horizontal orientation, this centers each of the individual lines. In addition, all Nodes in a single line will be aligned by their bottom, and the bottom line will be flush with the bottom of the Node. In vertical orientation, this bottom justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their center, and the layout will be centered in the Node.
CENTER	In horizontal orientation, this centers each of the individual lines. In addition, all Nodes in a single line will be aligned by their center, and the layout will be centered in the Node. In vertical orientation, this centers each of the individual columns. In addition, all Nodes in a single column will be aligned by their center, and the layout will be centered in the Node.

Anchor Result by Enumerator Name Cont.	
TOP_CENTER	In horizontal orientation, this centers each of the individual lines. In addition, all Nodes in a single line will be aligned by their top, and the top line will be flush with the bottom of the Node. In vertical orientation, this top justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their center, and the layout will be centered in the Node.
BOTTOM_RIGHT	In horizontal orientation, this right justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their bottom, and the bottom line will be flush with the bottom of the Node. In vertical orientation, this bottom justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their right, and the right column will be flush with the right of the Node.
MIDDLE_RIGHT	In horizontal orientation, this right justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their middle, and the layout will be centered in the Node. In vertical orientation, this centers each of individual columns. In addition, all Nodes in a single column will be aligned by their right, and the right column will be flush with the right of the Node.
TOP_RIGHT	In horizontal orientation, this right justifies all of the individual lines. In addition, all Nodes in a single line will be aligned by their top, and the top line will be flush with the bottom of the Node. In vertical orientation, this top justifies all of the individual columns. In addition, all Nodes in a single column will be aligned by their right, and the right column will be flush with the right of the Node.

Initializing from JSON

In order to specify that a Layout is a FloatLayout. The Format object for the node must be initialized in the json as type "Float".

The "format" block for FloatLayout takes the following fields:

- "orientation" (string) one of 'horizontal' | 'vertical'
    - "orientation" specifies whether this is a horizontally or vertically oriented FloatLayout.
- "x_alignment" (string) one of 'left' | 'center' | 'right'
    - "x_alignment", along with "y_alignment", specifies which type of anchor will be used by this float layout.
- "y_alignment" (string) one of 'bottom' | 'middle' | 'top'
    - "y_alignment", along with "x_alignment", specifies which type of anchor will be used by this float layout.

The "layout" block for a child within a FloatLayout takes the following fields:

- "priority" (int)
    - "priority" indicates the placement priority of this child. Children with lower priority go first.

Example JSON

```
"my_node": {
  "type": "Node",
  "format": {
    "type": "Float",
    "orientation": "horizontal",
    "x_alignment": "right",
    "y_alignment": "top"
  },
  "data": {
    ...
  },
  "children": {
    "child0": {
      ...
      "layout": {
        "priority": 0
      }
    },
    ...
  },
}
```

This example is taken from assets.json in the UIDemo.

GridLayout

Overview

GridLayout provides a grid layout manager.

A grid layout subdivides the node into equal sized grid regions. Each grid region may receive its own child (and can receive more than one). A grid region behaves like an AnchoredLayout for the rules of how to place the child.

Initializing from JSON

In order to specify that a Layout is a GridLayout. The Format object for the node must be initialized in the json as type "Grid".

The "format" block for GridLayout takes the following fields:

- "width" (int)
  - "width" specifies the number of columns in the grid.
- "height" (int)
  - "height", specifies the number of rows in the grid.

Each grid cell behaves as an anchored layout manager that always assumes percentage offsets. The "layout" block for a child within a GridLayout takes the following fields:

- "x_index" (int)
  - "x_index" indicates the horizontal grid index of this child.
- "y_index" (int)
  - "y_index" indicates the vertical grid index of this child.
- "x_anchor" (string) one of 'left' | 'center' | 'right' | 'fill'
  - "x_anchor", along with "y_anchor", specifies which type of anchor will be used for this child within its grid cell.
- "y_anchor" (string) one of 'bottom' | 'middle' | 'top' | 'fill'
  - "y_anchor", along with "x_anchor", specifies which type of anchor will be used for this child within its grid cell.

AnchoredLayout

Overview

AnchoredLayout provides an anchored layout manager.

An anchored layout attaches a child node to one of nine "anchors" in the parent (corners, sides, or middle), together with a percentage (or absolute) offset. As the parent grows or shrinks, the child will more according to its anchor. For example, nodes in the center will stay centered, while nodes on the left side will move to keep the appropriate distance from the left side. In fact, the stretching behavior is very similar to that of a NinePatch.

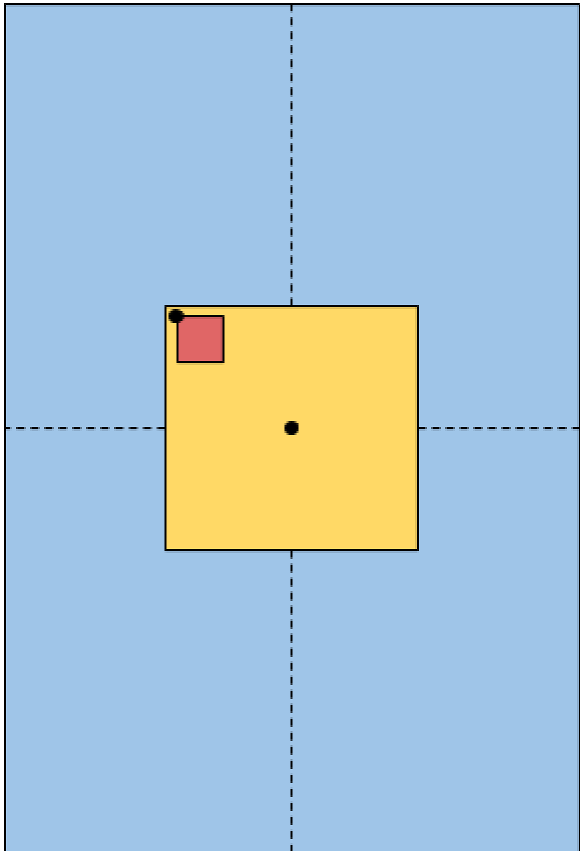
Initializing from JSON

In order to specify that a Layout is an AnchoredLayout. The Format object for the node must be initialized in the json as type "Anchored".

The "format" block for AnchoredLayout does not take any additional fields.

The "layout" block for a child within an AnchoredLayout takes the following fields:

- "x_anchor" (string) one of 'left' | 'center' | 'right' | 'fill'
    - "x_anchor", along with "y_anchor", specifies which type of anchor will be used for this child within its parent.
- "y_anchor" (string) one of 'bottom' | 'middle' | 'top' | 'fill'
    - "y_anchor", along with "x_anchor", specifies which type of anchor will be used for this child within its parent.
- "absolute" (boolean)
    - If "absolute" is set to true, "x_offset" and "y_offset" will be treated as an absolute offset in the parent's coordinate space rather than percentage offsets.
- "x_offset" (number)
    - "x_offset" specifies the horizontal offset from the anchor. If "absolute" is true, this is the distance in coordinate space. Otherwise, it is a percentage of the width.
- "y_offset" (number)
    - "y_offset" specifies the vertical offset from the anchor. If "absolute" is true, this is the distance in coordinate space. Otherwise, it is a percentage of the height.



Blue Box

Parent: None
Format Type: Anchored

Yellow Box

Parent: Blue Box
Format Type: Anchored
Data Anchor: [0.5, 0.5]
Layout X_Anchor: Center
Layout Y_Anchor: Middle

Red Box

Parent: Yellow Box
Data Anchor: [0, 0]
Layout X_Anchor: Left
Layout Y_Anchor: Top
Layout Absolute: True
Layout X_Offset: 5
Layout Y_Offset: 5

Figure 4: Here is the same scene as shown in Figure 1 using anchored layouts instead of strictly relying on node position and sizing. Notice that we no longer need the constants w_b, h_b, or h_y to determine the location of the Red Box. Additionally, this implementation will be aspect ratio and screen size agnostic, a must for any successful mobile application.

Using Widgets [Not applicable CUGL 1.2 or lower]

What are Widgets?

Widgets are sections of the scene graph that have been written in other files than that used for the main scene graph. These extra JSON files allow for pulling some of the boilerplate outside of the main assets.json (or whatever other main file is being loaded) by exposing named variables to be used in this main file that will then be mapped into internal variables of this now child scene graph. Widgets can be particularly useful when trying to encapsulate certain pieces of the scene graph into their own files, when using similar clusters of nodes in multiple places, or when hoping to improve readability by collapsing the boilerplate of more complex nodes (like buttons). Widgets can also contain other widgets and may be used to things as small as an individual node or as large as an entire scene.

Creating a new Widget in JSON

To create a new widget for use in the main asset manifest, first create a new JSON file. This file should have two top-level objects: “variables” and “contents”.

“variables” will define the named attributes that will be exposed by this widget to whatever scene graph definition choses to use it. Each variable is its own attribute and holds the path to the value that this variable should replace. These paths are encoded as string arrays. The “variables” block for a widget defining a basic play button might be as follows:

```
"variables" : {
  "anchor"   : ["data","anchor"],
  "scale"    : ["data","scale"],
  "x_offset" : ["layout","x_offset"],
  "y_offset" : ["layout","y_offset"]
}
```

The “anchor” variable for this widget will be mapped to the “anchor” attribute within the “data” block of the node encoded by this widget. Similarly, the “y_offset” variable will map to the “y_offset” attribute within the “layout” block. All widget variables are optional; they override existing default values.

The other block, “contents” contains the full definition of the node produced by this widget. This definition MUST include default values at all of the paths used in the “variables” section. Here is the “contents” block for the play button that uses the variables above.

```
"contents"  : {
  "type"   : "Button",
  "data"   : {
    "upnode"   : "up",
    "pushable" : [0,100,29,171,100,200,171,171,200,100,171,29,100,0,29,29],
    "visible"  : false,
    "pushsize" : true,
    "anchor"   : [0.5,0.5],
    "scale"    : 1.85
  },
  "children" : {
    "up"       : {
      "type"   : "Image",
      "data"   : {
        "texture"  : "play"
      }
    }
  },
  "layout" : {
    "x_anchor" : "center",
    "y_anchor" : "middle",
    "x_offset" :  0.035,
    "y_offset" : -0.175
  }
}
```

Notice that “contents” is a standard definition of a node as explained in the sections above. When this widget is used within the definition of a scene graph, the end result is equivalent to replacing the widget usage with this contents block. Note that the node defined here will not be named “contents” but rather match the name of the node defined by this widget in the main scene graph.

Using a Widget in the Scene Graph JSON

In order to use an externally defined widget in the scene graph, it must first be loaded as part of the "jsons" block (see the prior section on loading JSON data for more info on this). For the play button outlined above, that loading looks like this:

```
"jsons"  : {
  ...
  "play"   : "json/widgets/play.json",
  ...
}
```

Once the JSON that defines the desired widget has been loaded, it may be used to replace a node by using a special "Widget" node type.

The "data" block for any node defined with the "Widget" type takes the following fields:

- "key" (string)
  - "key" refers to the named JSON that has been imported earlier in this file.
- "variables" (JSON object)
  - "variables" provides values for all of the variables that have been exposed by the widget being used to replace this node. The fields of "variables" are entirely dependent on the widget being used and are always optional.

A usage of the play button widget defined above is written as follows.

```
"play"  : {
  "type"   : "Widget",
  "data"   : {
    "key": "play",
    "variables": {
      "anchor" : [0.5,0.5],
      "x_offset" :  0.035,
      "y_offset" : -0.175,
      "scale": 1.85
    }
  }
}
```

Changing one of the values passed in through the "variables" block will change the corresponding variable in this usage of the widget once the scene graph has been parsed and loaded. The values for these variables can be any JSON value of object. For example, the full "layout" block for the node within the widget might be replaceable as a variable.

Importing Custom Asset Types

Sometimes, the initial formats for imported data are not enough for the specificity required by a project. Even more likely, there is a custom JSON schema that encodes some arbitrary level, character, AI behavior tree, or other project specific type of object. For these cases, it may be useful to add an additional type that can be loaded in asynchronously to the AssetManager within CUGL. This section will walk through the basics of adding an additional loader for levels through a special case JSON parser. It assumes that there exists a Level class that is being loaded.

Step One - Add a new Loader Class

When adding a new asset loader, the first step is to create a new subclass of Loader<T> that is typed to the objects that this loader will load. For example, a LevelLoader that publicly inherits from Loader<Level>. Using the existing loaders as a guide, the key methods that this new loader needs (beyond constructors and the like) are read() (overridden from the base loader) and materialize(). This read() function will be called by the AssetManager when either synchronously or asynchronously loading these objects. After as much as possible has been done in a separate thread, the materialize() function will execute on the main thread to finish loading the new asset. This function is where the asset will be placed inside of the AssetManager to be accessed later by some key value.

Step Two - Attaching the Hook to the AssetManager

Now that the new loader exists, it is time to attach it to the AssetManager and have it handle any files imported as “levels.” To do this, the following line needs to be added before any assets of the associate type are loaded (note that this assumes a static alloc() method rather than a typical constructor):

```
_assets->attach<Level>(LevelLoader::alloc()->getHook());
```

This will associate the new loader with the Level class in the asset manager.

Step Three - Adding a Category to the Manifest

Now that the Level class is associated with an appropriate loader in the AssetManager, it is time to edit AssetManager::loadDirectory() and AssetManager::loadDirectoryAsync() to properly consider a section within the manifest a collection of Levels to be loaded. To do this, the following lines are added in keeping with the others already there:

```
[else] if (child->key() == "levels") {
  success = readCategory(typeid(Level).hash_code(),child) && success;
}
```

A similar block of code must be added to AssetManager::unloadDirectory() to ensure that the files are also properly unloaded.

Note that scenes are checked for separately than everything else because they must be loaded last. If adding a new type of asset that must load after other types, it should also be checked for separately between the larger list of assets and scenes.

With this named category now checked for specifically in the AssetManager, a new top-level object named “levels” can be added to assets.json (or another loaded manifest) and all levels listed there will be loaded asynchronously along with all fonts, textures, and sounds.

Some may find it useful to leave the read() and materialize() methods from Step One as stubs until completing Step Three to allow for debugging these type-specific methods.

Creating Custom Nodes

If the list of existing possible node types and widget options are insufficient for the needs of a particular project, it is both feasible and encouraged to create custom node types as needed. By creating a new subclass of Node, it is possible to override the default draw(), render(), doLayout(), updateTransform(), or other methods to alter behaviour and achieve new results.

Reasons for Creating New Nodes

Teams frequently create new custom node types for the following reasons:

- Custom Shader Work
- Custom Input Handling
- Complex Model Encapsulation
- Multi-Texture Rendering

The last of these is most commonly used when a team decides that they have enough dynamic objects being added and removed that it makes sense to have a part of their project simply represent a large drawing canvas with camera controls and other drawing methods similar to what they may be used to coming from other engines like LibGDX.

Adding Custom Node support to the SceneLoader

Aftering creating whichever custom nodes for the project, it is useful to add support for these new nodes to the scene graph parser. This requires making modifications to CUGL in a few key places.

1) Add a name related to the new node to the Widget enum in CUSceneLoader.h
2) Add a definition for this supported type with SceneLoader::init() in CUSceneLoader.cpp
3) Add a new case statement within SceneLoader::build() that calls allocWithData() for the new node

After adding these three additional pieces of code, it will be possible to use the new custom node "type" in the scene graph JSON with a "data" block passed into allocWithData() with any custom variables that are necessary to initialize the custom node. The initWithData() method in any custom node should first call the base Node::initWithData() to preserve functionality of the default "data" attributes available to all nodes.