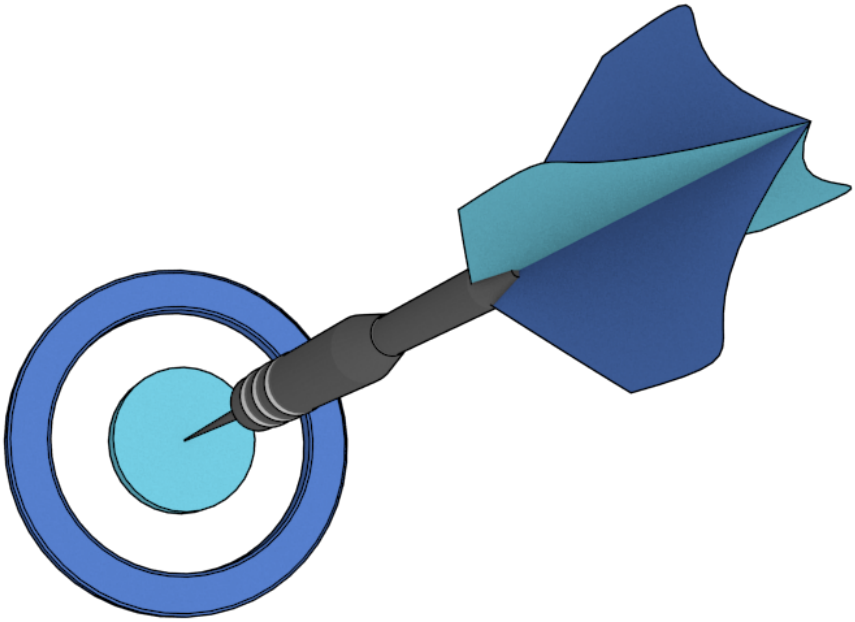
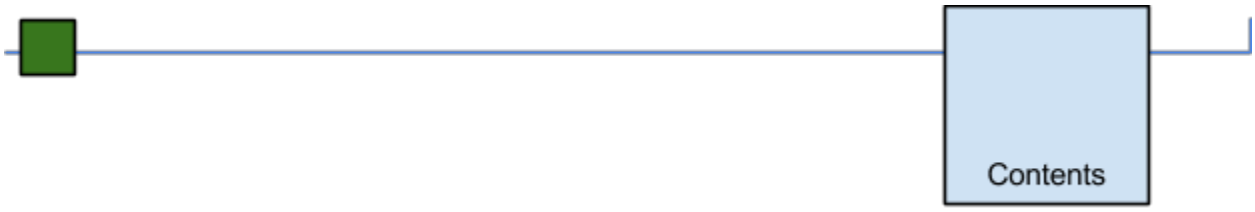
Ranger Scene graph

version 1.0.0

William R. DeVore



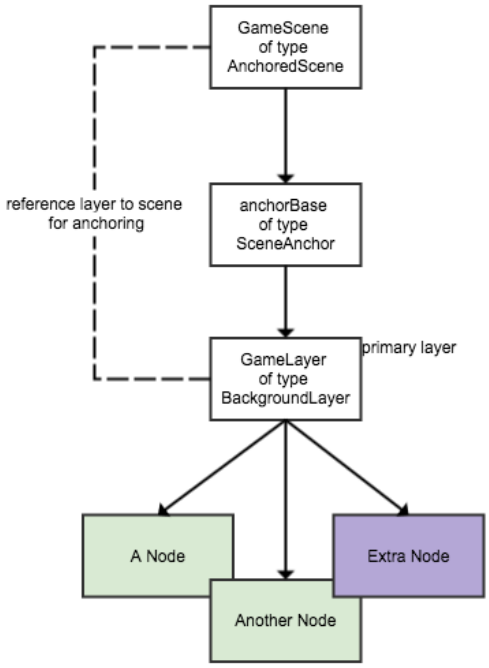


Scene graph

Internally **Ranger** maintains a scene graph of all the Nodes. The “root” of any active Scene is the Scene itself (aka GameScene).

The “real” root is a space called “world” which isn’t really part of the scene graph but is actually part of the transform stack.

In essence there are two types of stacks: Nodes and Transforms. The Node “stack” is what we see visually. The Transform stack is how the Nodes are spatially related.



AnchoredScene

AnchoredScene is a custom Scene designed for transitions. You are not required to use it, however, not doing so will limit your animations possibilities.

What AnchoredScene does is take your custom Layer and dock (Scene graph wise) it right below a SceneAnchor Node. This effective “isolates” your Layer from scene transitions. Your Layer has no idea it is being transformed when a transition is in progress. It is the Node that transitions are applied to.

SceneAnchor

SceneAnchor is a simple group node whose only value is in acting as a Rotation and Scale target. It is the Node that actually parents your Layer (aka GameLayer).

Background Layers

Ranger has a custom Layer called **BackgroundLayer**. It is designed to “fill in” the background and provide default input behavior. You don’t have to use it, just roll your own. Perhaps you don’t need input behavior, just copy BackgroundLayer and remove the input mixins. Any Node can act as a “layer” of a Scene. As a matter of fact you don’t even need a Layer. However, in general you will most likely use a BackgroundLayer for most Scenes.

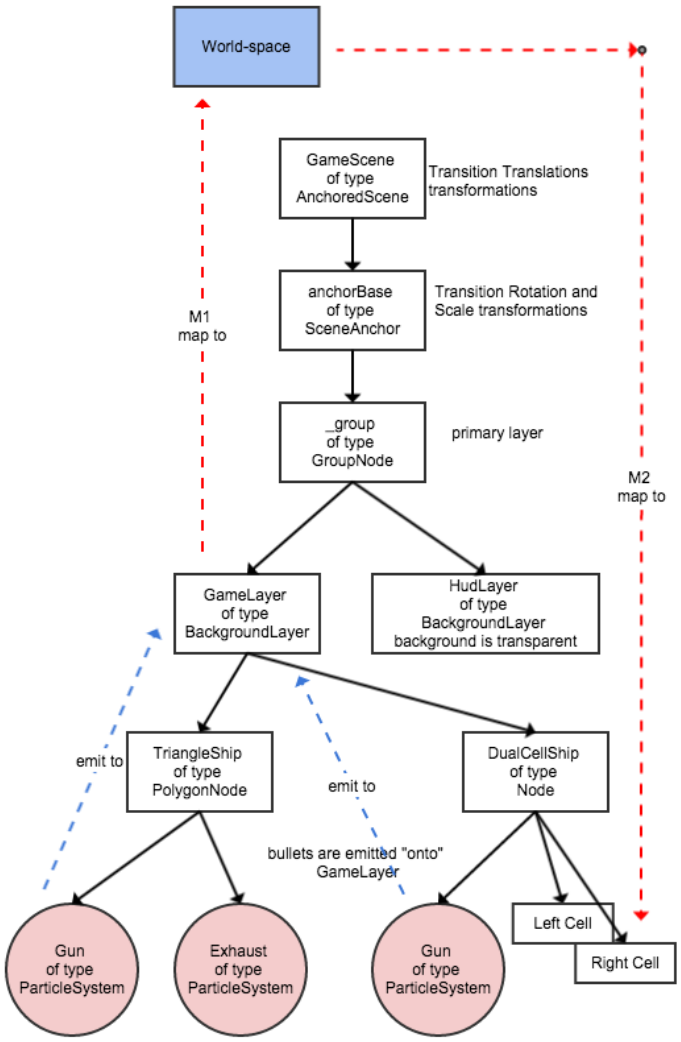
Grouping behavior

In order for a Node to contain children it needs to “mix” in the ***GroupingBehavior*** mixin. Once the behavior is mixed in a Node can begin collecting other Nodes. However, a vast majority of Nodes you create will most likely be Leaf Nodes (aka Nodes that don’t implement the GroupingBehavior mixin).

Mapping

With any type of Scene Graph there must be a way of mapping from the local-space of some Node to the local-space of another Node. This functionality is basic to every Node so much so that the BaseNode class is coded with many mapping methods.

Nodes relate spatially to each other by way of a transform, an Affine transform to be exact. At each level in the graph there is an Affine transform involved. In order to map a location from one Node to another we need a transform that can map your location to a common space and then map from that very same common space into a destination space.



This “common” space is called world-space. It is the space that is sandwiched between Device-Design space and the top most Node of the scene graph. This means that world-space is the “parent” space of all Nodes in the scene graph. This is what makes world-space the perfect intermediary for mapping between spaces.

Game engines that aren’t scene graph based typically leave all the hierarchical transformation work up to the developer; after all it is the developer who understands their requirements. This amounts to a single-space game layer. The upside is that the code is more concise and focused leading to a

better performance, however, the code is certainly more complex. In the long run the developer isn't escaping the need to map from space to space they are just doing it in a more integrated fashion.

With scene graphs we have the transformation information hierarchically tracked by way of their placement within the graph. We can ask the scene graph to map from one to another via world-space.

Referencing the above diagram, let's say we want to test the for the collision between the TriangleShip's bullet and the "Right Cell" of the DualCellShip. In RangerRocket bullet particles are emitted "onto" GameLayer space. We do this because once a bullet leaves the barrel it no longer is a "part of" the ship. This makes sense in both the real world and the game.

So we need to map from GameLayer space to the Right-Cell space. We do this via world-space. First we take the bullet and map it to world-space (M1)

```
...
      if (p.active) {
        Ranger.Vector2P pw = p.node.convertToWorldSpace(_localOrigin);

        collide = _dualCellShip.pointInside(pw.v);
...
```

then inside the *pointInside* method we map the point into the RightCell's local-space (M2)

```
...
        nodeP = _rightCell.convertWorldToNodeSpace(point);
        collide = _rightCell.pointInside(nodeP.v);
...
```

Finally we perform a point-inside-polygon test.

This is how scene graphs function in **Ranger**.