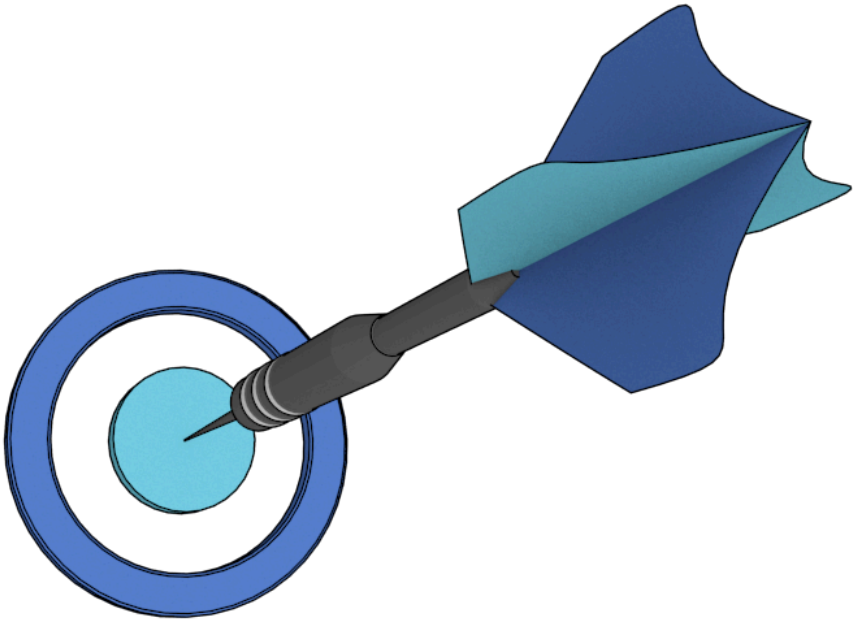
Ranger Nodes

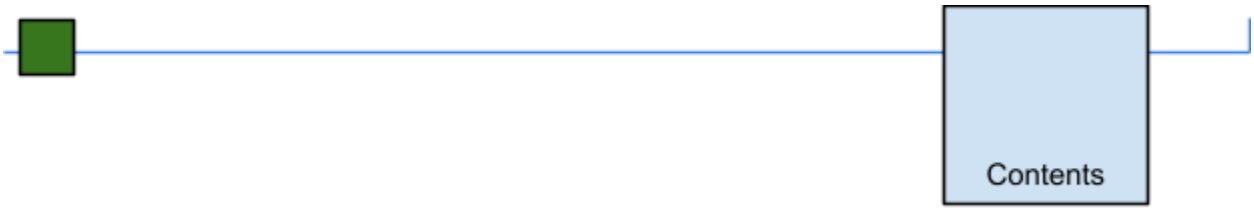
version 1.0.0

William R. DeVore



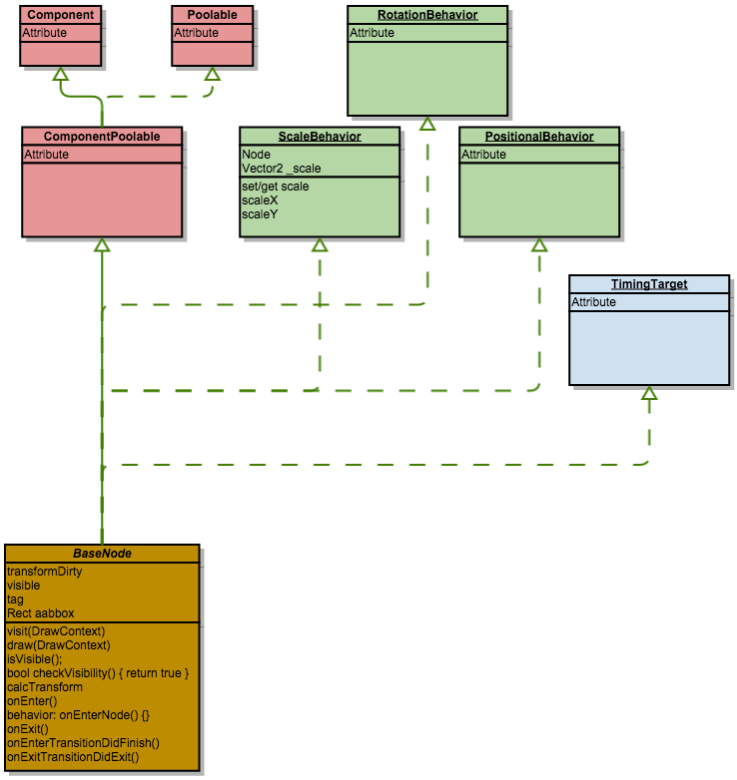
Ranger Nodes

Ranger version 0.1.x



BaseNode

The very bottom of the Node tree is the abstract class *BaseNode*. This class provides four basic features: Pooling, Transforms, Mappings and Timing; see below.



Pooling

Ranger's pooling is based on Dartemis's pooling and it is completely optional to participate in. Any time you elect not to call a Node's *moveToPool*() you are basically saying "I don't care if the garbage collector (GC) collects it". But if you do call *moveToPool*() then the pool will hold a reference to it thus it keeping from the prying eyes of the GC. Of course the flipside is that you are consuming memory. So be conscious of how you use pooling. Some Scenes are temporary and likely never to be seen again during the lifetime of the game. In that case you can probably forgo pooling, again simply forget to move the object back to the pool.

Here is a code snippet that occurs in the Ranger-Rocket app where we are performing collision detection between the ship's bullet and one of the shapes:

```
        Ranger.Vector2P pw = p.node.convertToWorldSpace(_localOrigin);
        Ranger.Vector2P nodeP = _squarePolyNode.convertWorldToNodeSpace(pw.v);

        collide = _squarePolyNode.pointInside(nodeP.v);
```

```
        if (collide) {
          _handleBulletToSquareCollide();
          pw.moveToPool();
          nodeP.moveToPool();
          return p;
        }

        pw.moveToPool();
        nodeP.moveToPool();
```

We need to map the bullet that is in GroupNode space into the local space of the Node where we want to check for a collision. Once the check is complete we don't need the pooled versions of the Vector2 objects any longer so we "move them" back to the pool. I chose to move them to the pool because on the very next "update" I am going to perform the very same check again. So it makes sense to "reuse" pooled objects rather than giving the GC a colossal migraine; and when the GC has a migraine you will feel the pain too as your game "pauses" while the GC takes its revenge.

Here is another scenario. Let's say that when your bullet hits a square Node it explodes and disappears never to be seen again. Your code will probably call removeChild(child, **true**). Pay attention to the last parameter. In this case **Ranger**'s scene graph will detach the child AND move the square Node back to the pool. The question is "when do you use **true** or **false**"?. It all depends on your game. Perhaps squares will appear unending and your player's task is to destroy them forever; in this case it may be prudent to use "true".
If it is a Node that will never appear again and memory is of no concern then "true" is also a valid choice. But if resources are scarce then probably "false" is better and you just have to deal with the GC. The less work the GC has to do the less noticeable the pause will be.

Transforms

All Nodes have three basic transforms: Translate, Scale and Rotate.

Skew
Skew is a separate transformation that I felt wasn't used that often. However, to have skewing functionality just "mixin" the **SkewBehavior** class.

Translate

To translate a Node you would call one of several methods from the **PositionalBehavior** mixin. You could set the position based on a Vector2 or by component:

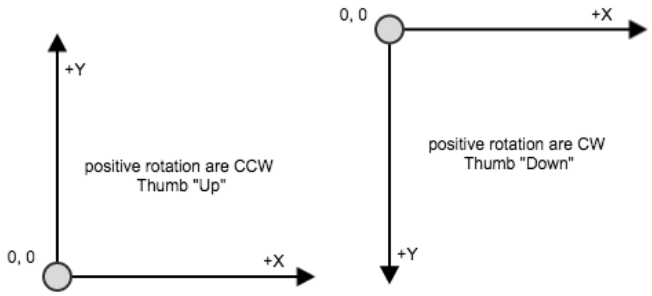
Rotate and Base coordinate systems

You can rotate a Node by either Radians (node.rotation = 3.14) or Degrees (node.rotationByDegrees = 45.0) using the **RotationalBehavior**'s methods.

The question is which “way” will it rotate, Clockwise or Counter Clockwise. That depends on the orientation of the base coordinate system which is defined in *config.dart*:

```
static const bool base_coordinate_system = LEFT_HANDED_COORDSYSTEM;
```

I arbitrarily choose the nomenclature of “left/right” even though it really applies to 3D space. It is more like: left-handed means the left hand’s thumb is “up” and right-handed means that the left hand’s thumb is “down”:



The default is the “thumb Up” orientation. In this configuration the origin is in the lower-left corner of the view; unless you center the Node upon construction.

Scale

You can scale a Node uniformly (node.uniformScale = 2.0) or non-uniformly (node.scaleX = 2.0) by using ScaleBehavior’s scaling properties. Scaling occurs around the Node’s local origin.

Space Mappings

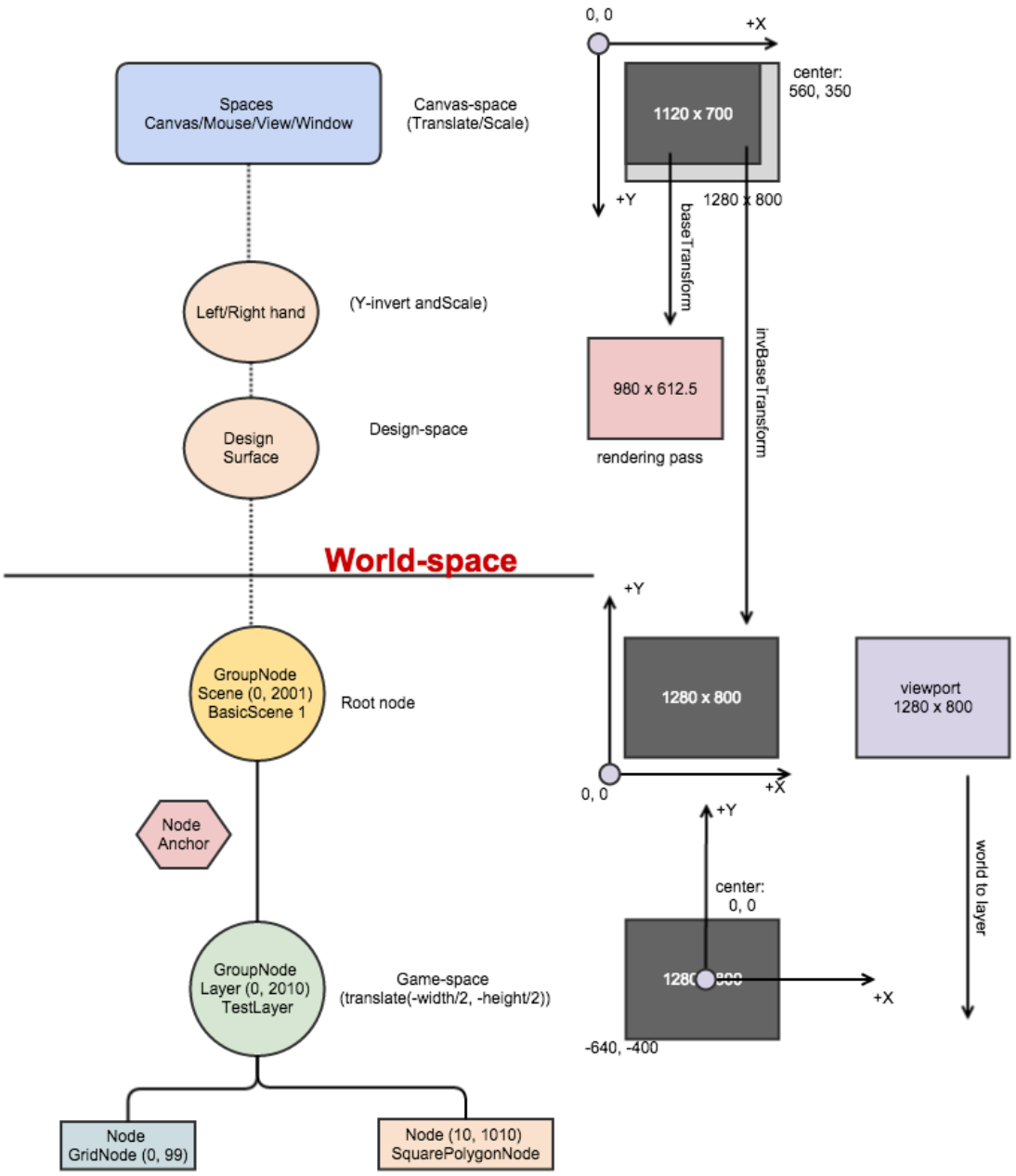
BaseNode also provides mapping methods for moving from one Space to another.

```
Ranger.Vector2P pw = p.node.convertToWorldSpace(_localOrigin);
Ranger.Vector2P nodeP = _squarePolyNode.convertWorldToNodeSpace(pw.v);
```

Above we are taking a local-space coordinate--in this case (0, 0)-- and first mapping it to world-space and then mapping that into the local-space of the square. So what is “world-space”?

World-space

World-space is the space that bridges Canvas/Design region to the scene graph region; its vaguely the “root” of the scene graph. You can view world-space coordinates but they will seem a bit “skewed” and distorted depending on the ratio of Design-to-Device resolutions.



World-space is not a good space to be in but it is a good space to pass through. Whenever you want to map from one Node to another you do so via World-space and that is because world-space is a common space for all Nodes.

Timing

Routing timing to your classes (aka targets) is it pretty easy. **Ranger** has a Scheduler that can handle two types of “targets”: Functions of type

```
typedef void UpdateTarget(double dt);
```

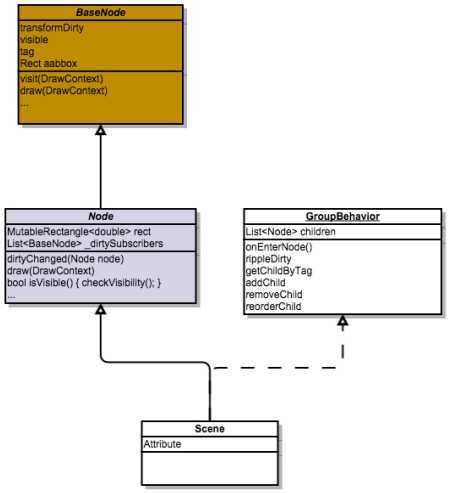
and ***TimingTarget***s. Each can be used to route timing to your classes.

TimingTarget type classes are used quite frequently. The **TweenAnimation** class, BaseNode, **Sprites**, **Particle Systems**.

It is pretty easy to setup a class to receive timing. Just inherit from TimingTarget and schedule it with the Scheduler. From there on out your class will receive timing data delivered to your update() method until you unschedule it. Here is an example of a particle system being scheduled:

```
ps = new Ranger.ModerateParticleSystem.initWith(maxParticles);
...
app.scheduler.scheduleTimingTarget(ps);
```

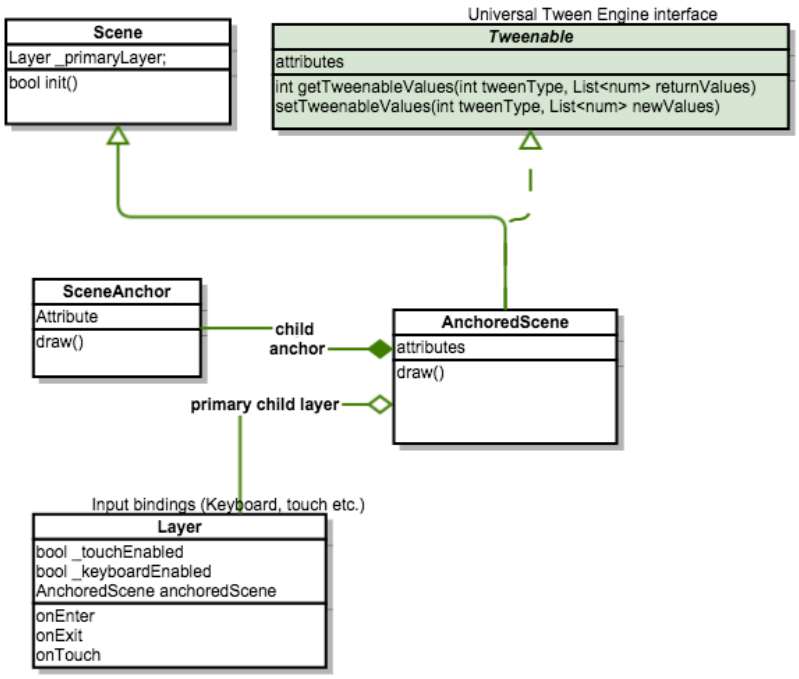
Scene

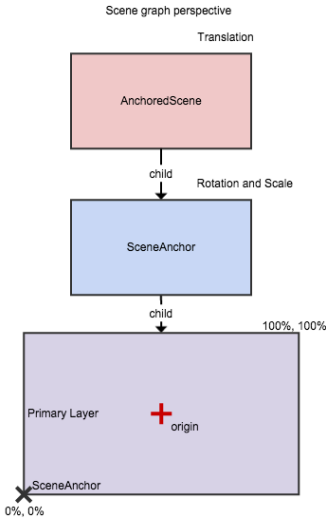


A Scene is a container for other Nodes because it “mixes” in the *GroupBehavior* mixin. Most of the time Scenes will contain only Layers.
The inverse is if you create Nodes that don’t mixin grouping behavior then what you have created is a Leaf Node. A **TextNode** is an example of a leaf Node.

Most often a Scene will have a 1-to-1 correspondence with Layer where the Layer supplies the Nodes that are visible. Scenes for the most part are “transparent”.

There is one very special Scene called BootScene. This scene is incredibly simple. It simply waits for **Ranger** and Dartium to finish bootstrapping then immediately replaces itself with a replacement scene, which in most cases in a Splash scene.



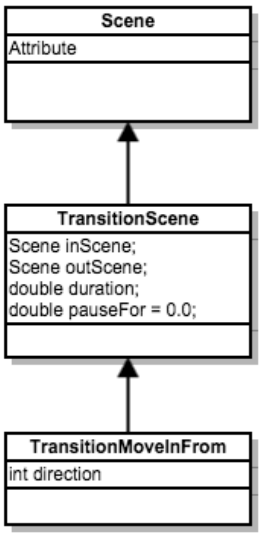


AnchoredScene

The most important scene is the **AnchorScene** and is also the Scene that just about any scene you create will inherit from. Above is the inheritance structure of an AnchoredScene. We see that it is animatable because it implements the ***Tweenable*** interface, doing so allows the Scene to be directly animated by the Universal Tween Engine (UTE). We also see that it creates a **SceneAnchor** Node internally and then “binds” your “primary” layer as a child to the anchor; see image on the right. This creates a perfect arrangement where we can translate, scale and rotate Scenes all without disturbing the primary layer (aka the layer you created).
It is the AnchoredScene that gives Transitions their ability to function properly.

Transitions

A Scene is also used for is transitions. Transitions operate solely on Scenes and they themselves are Scenes. But what manages the scenes and their transitions? The SceneManager.



SceneManager

The SceneManager (SM) controls and managers scenes. Internally the SM maintains a “stack” and the Scene at the top of the stack is the scene that is running. During transitions the Outgoing scene is “stopped” and the Incoming scene is “started”.

When you start coding your own transitions one thing to be aware of is that the incoming scene becomes active immediately which means its

onEnter() method is called, which also means that the incoming scene will become visible. Looking at a code snippet of the **TransitionMoveInFrom** class we can see that the incoming scene “inScene” has its position placed completely out of view:

```
    switch (_directionFrom) {
      case FROM_LEFT:
        inScene.setPosition(-Application.instance.designSize.width, 0.0);
```

Once that is done an animation is created to “move it” back into view thus creating the illusion of a transition.

The SM also provides methods for pushing, popping and replacing Scenes.

Push

Pushing a scene causes the currently running scene to pause while activating the scene just pushed onto the scene stack.

Replace

Replacing a scene causes the currently running scene to be destroyed and removed while activating the new scene and placing it on the stack.

Pop

Popping a scene causes the currently running scene to be destroyed and removed while activating the next scene available on the stack. If there are no more scenes then **Ranger** pretty much stops and shuts down.

Finally the SM’s main job is to “visit” the scene supplying a DrawContext. Some Nodes may perform a visible check. If the Node determined it wasn’t visible the draw() method is skipped.

Layers

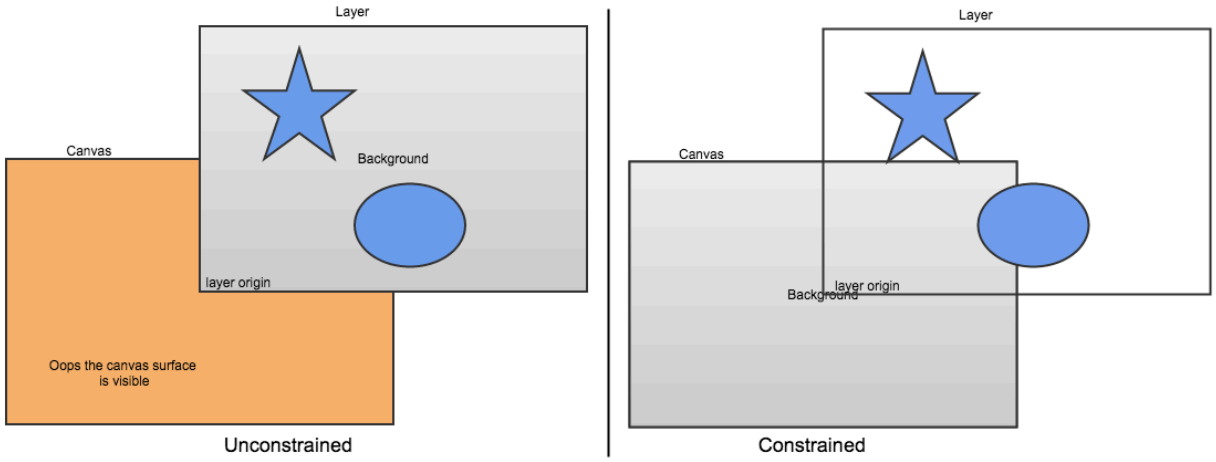
Layers are grouping Nodes that provide a good place to put visual Nodes and code logic. They pretty much accompany every Scene. Of course there are exceptions, for example BootScene has no Layer, and none of the Transition scenes because none of them have anything to show.

BackgroundLayer

The most important layer that comes with **Ranger** is the **BackgroundLayer**. This Layer “pre”-mixes all the currently supported Inputs: Mouse, Keyboard and Touch. It also has another trick up its sleeve, it can constrain the background positional wise. This insures that the background always fills the viewport no matter where the Layer is translated to.

Setting constrainBackground = false will cause the Layer’s background to “stick” with the Layer’s origin which isn’t good if the Layer has been moved (for example, centered), in addition,

the actual Canvas surface would be exposed (it is Orange by default). So if you are seeing a lot of Orange then you have exposed the surface--not really good. Take a look at the image below.



On the left the background isn't constrained so when the Layer moves so does the background, which exposes the surface. Most of the time you don't want to expose the surface which is why the default is to constrain the background to the view.

Hint: with multiple layers only the bottom most layer should to be constrained.

OverlayLayer

This Layer is almost identical to the BackgroundLayer but it doesn't have a constrained background and defaults to a transparent background. You would typically use this as a Head-Up-Display (HUD).

Of course you are free to create your own custom layers and encouraged to do so.

Cascade and Multiplex

Cascading involves cascading a single color (including alpha) from parent down to child. This allows for fading in and out on entire scenes as well as individual Nodes.

Multiplex -- not implemented.

Nodes

Node provides a default implementation of BaseNode plus the ability to handle subscribers looking for notifications when Nodes have become dirty.

There are several other auxiliary Nodes that can come in handy.

EmptyNode

EmptyNode is great as a non-visual placeholder left type node--of course you can always enable its visual to see a placebo.
A good example can be seen in Rocket-Ranger. EmptyNodes are used as targets for the ships' exhausts. When the ship is transformed the exhaust (aka _exhaustLeftPort node) changes relative to the GameLayer (or a grouping layer perhaps) where the particles are to be emitted. On every update() the exhaust particle system's position is updated with the position of the EmptyNode mapped into GameLayer-space. By setting the position of the particle system your are in effect dictating where the next particle is emitted.

```
    Vector2 gs = _convertToGameLayerSpace(_exhaustLeftPort.position);
    _exhaustLeftPS.setPosition(gs.x, gs.y);
```

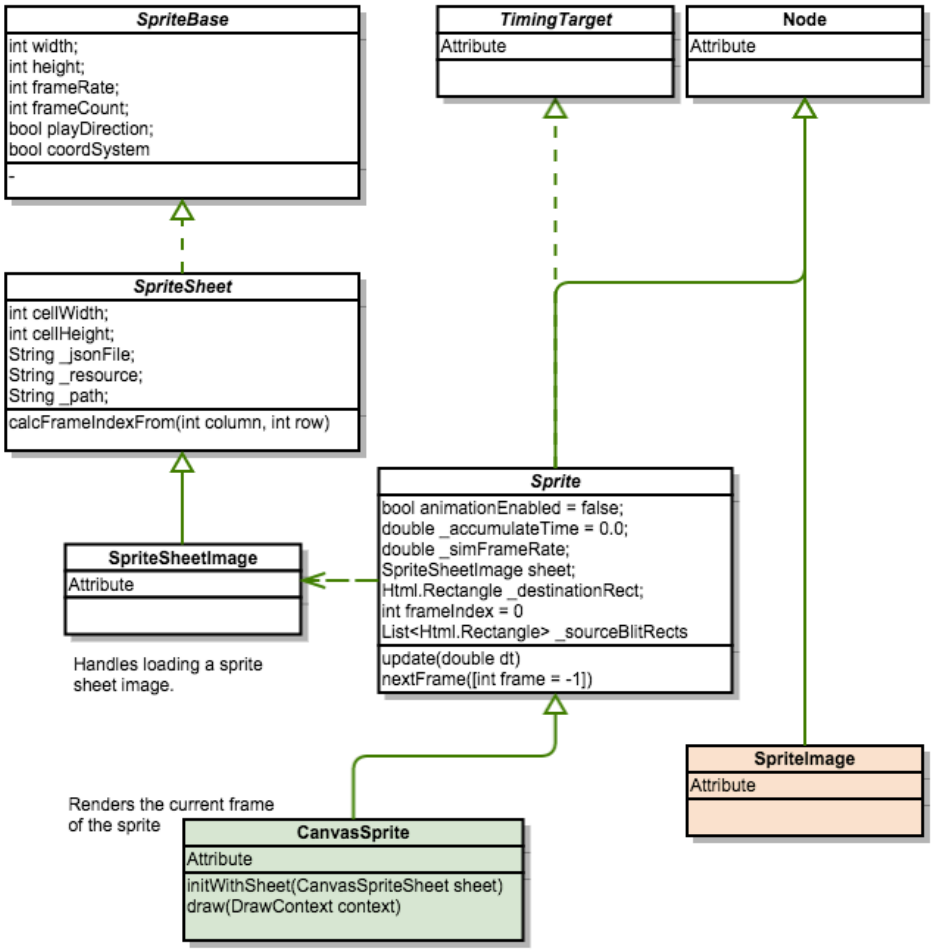
GroupNode

GroupNode is also a non-visual Node. Its main job is simply to contain other Nodes.
A good example can be seen in Rocket-Ranger. A GroupNode is used as the primary layer so that more than one Layer can exist in AnchoredScenes. In this case both a GameLayer and HudLayer are created.

```
    _group = new Ranger.GroupNode();
    _group.tag = 2011;
    initWithPrimary(_group);

    _controlsPanel = new ControlsDialog.withHideCallback(_panelAction);

    ...

    //-------------------------------------------------------------
    // Main game layer where the action is. ddddaa = olive green
    //-------------------------------------------------------------
    _gameLayer = new GameLayer.withColor(Ranger.color4IFromHex("#666666"), true);
    addLayer(_gameLayer, 0, 2010);

    //-------------------------------------------------------------
    // A layer that overlays on top of the game layer. For example, FPS.
    //-------------------------------------------------------------
    _hudLayer = new HudLayer.asTransparent(true);
    addLayer(_hudLayer, 0, 2012);
```

Sprites

There are two types of sprites: single frame(**SpriteImage**) and multi-frame (**CanvasSprite**).



SpriteImage

A SpriteImage is the Node representation of an ImageElement. First you would load the ImageElement then create a SpriteImage. The ImageElement should be loaded asynchronously and upon loading create your SpriteImage Node.

Normally you create a Resource class that handles the asynchronous loading by returning a Future. **Ranger** comes with an asynchronous **ImageLoader** class to assist your Resource class. The ImageLoader can simulate performance delays by artificially injecting a Future delay using Future.delayed.

```
class Resources {
...
```

```
  Future<ImageElement> loadImage(String source, int iWidth, int iHeight, [bool
simulateLoadingDelay = false]) {
    Ranger.ImageLoader loader = new Ranger.ImageLoader.withResource(source);
    loader.simulateLoadingDelay = simulateLoadingDelay;
    return loader.load(iWidth, iHeight);
  }
...
}
```

Placebo

However, you may notice in the examples and unit tests a “placebo”. Placebos are lightweight temporary “stand-ins” that, optionally, you can use immediately--no downloading required. They are typically embedded Base64 encoded resources. An embedded resource guarantees immediate availability and is excellent visual while the actual image is downloading.

Ranger comes with two embedded SVG images: spinner and spinner2. These types of resources are easy to create. Just find a website (or create your own converter using Dart) that converts text into Base64 encoded strings--the text in this case is an SVG definition. One such site is: MobileFish, it was used to create both spinners.

TODO: Base64 encoder app
We should create a simple Base64 encoder utility for Ranger-Sack. Dart has the encoder, we would just need to create a web app using it.

To use a placebo just create a SpriteImage from an embedded resource:

```
      Ranger.SpriteImage placebo = new Ranger.SpriteImage.withElement(resources.spinner);
      addChild(placebo, 10, 7000);
      // Track this infinite animation.
      app.animations.track(placebo, Ranger.TweenAnimation.ROTATE);

      UTE.Tween rot = app.animations.rotateBy(
          placebo,
          1.5,
          -360.0,
          UTE.Linear.INOUT, null, false);
      //                 v---------^
      // Above we set "autostart" to false in order to set the repeat value
      // because you can't change the value after the tween has started.
      rot..repeat(UTE.Tween.INFINITY, 0.0)
         ..start();
```

In the example above we add the “spinner” resource as a child, and we also want the placebo animated so we attach a Tween animation to it too.

Now we can begin loading the actual image while the placebo animates,

```
resources.loadImage("resources/grin.svg", 32, 32, true).then((ImageElement ime) {
  // Image has finally loaded.
  // Terminate placebo's animation.
  app.animations.flush(placebo);

  // Remove placebo and capture index for insertion of actual image.
  int index = removeChild(placebo);

  // Now that the image is loaded we can create a sprite from it.
  _grin = new Ranger.SpriteImage.withElement(ime);
  _grin.uniformScale = 5.0;
  // Add the image at the place-order of the placebo.
  addChildAt(_grin, index, 10, 101);
});
```

and once it loads we stop/flush the placebo's animation and remove it.

All this is done inside an immediately invoked Closure. The closure allows us to "capture" the placebo within local scope without having to create a class scoped variable. To me closures are kind of like "pictures in time" or "snapshots of a moment".

CanvasSprite

CanvasSprite is a simple implementation of a sprite sheet animator. It is almost certain you will create your own sprite sheet class.

CanvasSprite is similar to a SpriteImage in that it is driven by a single image, but the image is actually a series of smaller images compacted together in some form of a grid. Typically a JSON file accompanies the image. This associated file indicates how each smaller image is placed "on" the sheet image and what the frame-rate should be.

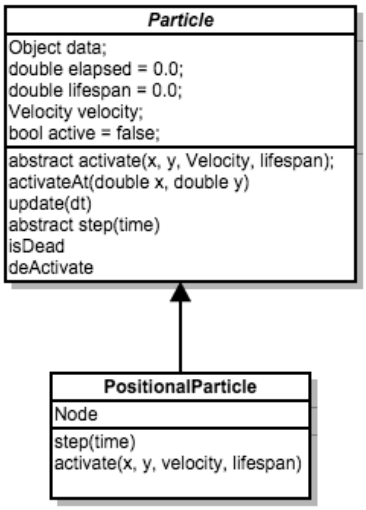
Because CanvasSprite implements the TimingTarget interface it can be registered with the Scheduler. Once scheduled the sprite receives timing data in the form of fractional seconds. **Ranger** only receives "clock ticks" at a resolution of ~1/60 of a second which means simple math is needed to calculate slower rates. Faster rates result in frame skipping.

Particles and Particle systems

Particle systems are infinite in nature. It is literally impossible for **Ranger** to provide the end-all-to-be-all particle system; even the RangerParticles app had to draw a line. Nonetheless, **Ranger** does supply a basic starter structure for particle systems, and it comes with two: **BasicParticleSystem** and **ModerateParticleSystem**--hopefully more advanced systems will show up in Ranger-Sack.

Particles

Particles function off of one basic premise: lifespan, anything more than this is icing on the cake. **Ranger**'s Particle class provides that plus **Velocity** for extra measure.



But there is more to "life" than just lifespan. The most basic particle is one that moves (PositionalParticle). It is pretty boring but it functions.

Considerably more complex particles are **TweenParticle** and **UniversalParticle**. They provide examples of Rotation, Scale and Color behaviors, and they do it completely different from each other. TweenParticle uses the Universal Tween Engine while UniversalParticle uses basic LERP.

But none of these particles are visible, they are just behaviors. Visibility is added by assigning a Node to the PositionalParticle during construction. RangerRocket has an example of this in the _populateParticleSystemWithCircles method, here is a snippet:

```
    CircleParticleNode protoVisual = new CircleParticleNode.initWith(Ranger.Color4IBlack)
          ..visible = false
          ..uniformScale = 1.0;

    Ranger.UniversalParticle prototype = new Ranger.UniversalParticle.withNode(protoVisual);
```

The UniversalParticle is given a **CircleParticleNode** to act as its visualization. But the code looks a bit odd. Why are the particle components implying they are "prototypes"? This is where Particle Systems come in to play.

Particle Systems

You construct Particle Systems (PS) with two components; a prototype visual and a prototype particle.

```
    ps.addByPrototype(_gameLayer, prototype);

    // The prototype is no longer relevant as it has been cloned. So
    // we move it back to the pool.
    protoVisual.moveToPool();
    prototype.moveToPool();
```

The PS takes the prototypes and clones them according to how many particles were specified during the creation of the PS. Shown above, once the prototypes have been cloned they are no longer needed and can be moved back to the pool. It is also permissible to NOT put them back

in the pool as they will never be needed again plus it would free up resources, use your own judgement.

Once you have a PS built your PS can start activating individual particles

```
gunPS.activateByStyle(Ranger.ParticleActivation.UNI_DIRECTIONAL);
```

or all the particles at once

```
_contactExplode.explodeByStyle(Ranger.ParticleActivation.OMNI_DIRECTIONAL);
```

Activation

Particle systems are not coded to activate a particle in any specific way as there is an infinite number of ways to activate them. Instead you provide a **ParticleActivation** class that can generate the configuration for each particle prior to activation.
Ranger comes with two example activators: **SimpleParticleActivator** and **RandomValueParticleActivator**. You allocate one, fill out a bunch of variance values and then assign it to a PS

```
    Ranger.RandomValueParticleActivator pa =
_configureForExhaustActivation(Ranger.Color4IRed, Ranger.Color4IYellow);

    _contactExplode.particleActivation = pa;
```

With some clever variance values RandomValueParticleActivator can produce some crazy effects. Both activators barely scratch the surface of what you can do.

Timing

Finally you need to route timing to your newly created awesome PS. You have two ways you can do that. The first only works if your Layer has scheduled updates for itself (which means it has called the scheduleUpdate() in the onEnter method and overridden the update() method). If that is the case then all you need to do is manually call the PS's update from within your Layer's update

```
_contactExplode.update(dt);
```

The slight advantage to this is that you don't need to remember to unschedule your PS.

Option #2 is to actually register your PS with the Scheduler.

```
Ranger.Application.instance.scheduler.scheduleTimingTarget(_contactExplode);
```

You can do this because PSs implement the TimingTarget interface. However, you will need to unschedule it in the onExit() method of your Layer

```
@override
void onExit() {
  super.onExit();
  Ranger.Application app = Ranger.Application.instance;
  app.animations.stop(_listSprite, Ranger.TweenAnimation.ROTATE);

  Ranger.Application.instance.scheduler.unScheduleTimingTarget(_contactExplode);

  unScheduleUpdate();
}
```

Failing to unschedule your PS will simply result is wasted cycles as your PS is feed timing data uselessly; generally not good.

End.