JavaScript Scene Graph Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

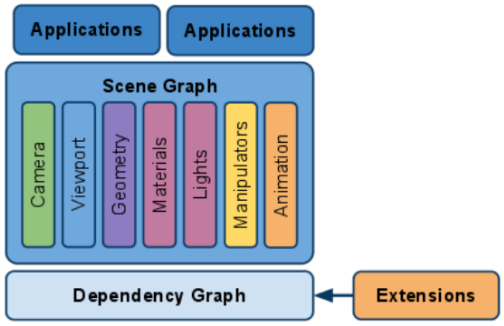
Chapter 1. Introduction

1.1. Installation

Installation instructions can be found here: “JS Developers Install Guide” [http://documentation.fabric-engine.com/latest/FabricEngine-JSDeveloperInstallGuide.pdf]

1.2. What's Fabric Engine's SceneGraph?

The **SceneGraph** is a wrapper for its counterpart, the Fabric Engine Core. Fabric Engine's core is a very low level system, providing the core objects such as a **Dependency Graph Node**, **Operators** and other elements. The **SceneGraph** however is an abstraction layer for the core. It provides *presets* for Dependency Graph Node setups, called the SceneGraphNodes. The SceneGraph is purely implemented in JavaScript, and therefore can be customized easily for any specialized purpose.



The SceneGraph is a factory which is used to create SceneGraphNodes. Each type of SceneGraphNode is registered with the SceneGraph on load. The factory function is responsible for constructing the underlying dependency graph, and returning an public interface. All data associated with the scene graph node is contained in the closure defined by the factory function.

A good understanding of closures and how they work in JavaScript is essential for understanding the Fabric SceneGraph.

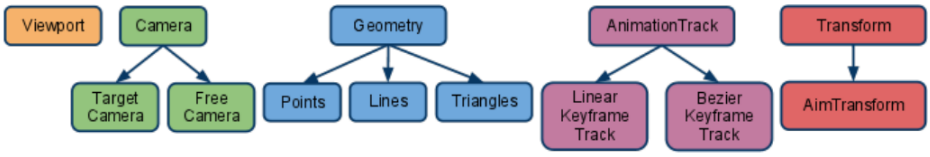
The SceneGraph is provided as a series of JavaScript files which implement factory functions for a certain usage field, for example the file *Images.js* contains several factory functions implementing 2D images, 3D images and video SceneGraphNodes.

SceneGraphNodes encapsulate the core objects necessary to provide a certain functionality. The VideoNode, for example, contains the Dependency Graph Nodes to store all of the video related data, as well as all of the Operators to read and manipulate the video stream. Moreover it provides JavaScript functions to access and manipulate the video.

Managers are similar to SceneGraphNodes and are used mainly to provide utility functionality. One example of a manager is the UndoManager. Managers are genrally singletons, in that only on of each type of Manager is constructed.

The SceneGraph also provides graphical utilities, such as the **Debugger**, which provides an graph based representation of the constructed core Dependency Graph. Other tools, for example the **Curve Editor**, provide a graphical Curve editor for keyframe animation curves.

1.3. Abstraction of the Dependency Graph



The goal of the SceneGraph layer is to present an interface to developers focused on SceneGraph construction. The interfaces exposed follow common SceneGraph conventions.

SceneGraphs can be setup in a few lines of code, and provide a rich array of features, from shader management and rendering, to manipulation and animation.

The SceneGraph is written entirely in JavaScript, and it abstracts the notion of the dependency graph away from the user. The JavaScript code assembles a lower level dependency graph.

1.4. Scene Object and SceneGraphNode Construction

The **scene** provides basic services such as time management, and and is used to construct SceneGraphNodes.

```
var scene = FABRIC.SceneGraph.createScene();
```

The first SceneGraph object to be constructed is usually the scene object. The scene object creates a Fabric Engine context, a dependency graph node for containing globals, and some event handlers that are used in rendering. It returns a public interface which provides methods for constructing SceneGrapNodes.

All SceneGraphNode types implemented in the SceneGraph are registered by providing their type as well as a constructor function. A SceneGraphNode can be constructed iin the following way:

```
var sgNode = scene.constructNode('Image2D', { name: 'myImageNode' } );
```

The construction options supported by each SceneGraphNode can be found in the **SceneGraphNode Reference** of this document.

1.5. Public and Private Interfaces

Each SceneGraphNode provides a private as well as a public interface. The public interface provide the public interface for the node. Outside of constructor functions it's only possible to access the public interface, while inside of constructor functions it's possible to access both the public and the private interface. This private interface enables exposing functions to tother nodes that request access to the private interface. Private interfaces expose methods that are used when building the underlying dependency graph. The public interface of a node should expose functions which define very high level behavior. For example, the sphere primitive exposes a public interface with methods to get and set the radius, and the private interface exposes methods to retrieve the core DGNodes that contain all the vertices. The public interface is what would be used to populte a graphical user interface for a node. Since the options used for the construction of a node can contain references to a public interface, you can retrieve the private interface by using this call:

```
var privateInterface = scene.getPrivateInterface(publicInterface);
```

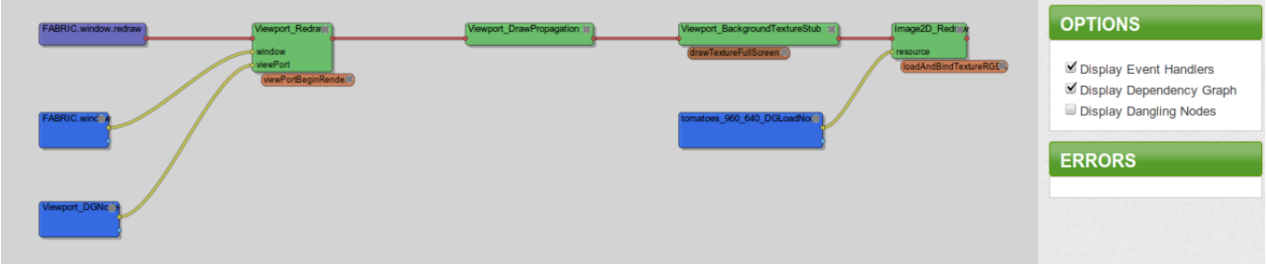

To access the public interface from a private one, you simply need to access the *.pub* member of the private interface. Managers can also access private interfaces, allowing them to perform changes to the private data, for example.

```
var sgNode = {
  pub: {
    publicMethod: function () {}
  },
  privateMethod: function () {}
};
```

1.6. Custom SceneGraphNodes and Inheritance

Applications can register their own, custom SceneGraphNodes. This can be useful and necessary when extending an existing node or providing a completely new one. In the constructor function of the custom node the other nodes can be accessed through their private interfaces, allowing access the inner workings of the SceneGraph. Each Scene Graph node extends the private or the public interface of its parent node type, enbling inheritance through composition. It is also possible to override an existing function on each interface by simply setting it to different function inside the custom node's constructor. For further details refer to the tutorials sections of this document.

1.7. The SceneGraph Debugger



The debugger is a useful tool to inspect the dependency graph contructed by the SceneGraph. Especially when building custom SceneGraphNodes this can be very helpful. The debugger is described in more detail in the section on the Drawing Pipeline Guide in this document. You can open the debugger by using executing this code in JavaScript:

```
FABRIC.displayDebugger();
```


Chapter 2. SceneGraph Type System

Fabric Engine enables you to define your own data structures. All data types in Fabric are define dynamically as part of the loading process of your application. To define the structure of data, you need to introduce **Types**. Types are very similar to structs in C++. Core types, such as **Integer**, **Boolean** or **String** are defined by the core. More complex types, such as the **Vec3**, for example, are defined by the SceneGraph.

2.1. Querying types

You can access all of the currently registered types, by calling on the **RegisteredTypesManager**. The manager is only accessible in the scope of a SceneGraphNode factory function, but you can access a dictionary of all current types by accessing the **RT** object:

```
console.log(FABRIC.RT);
```

Types are typically implemented in separate JavaScript files. The ones provided by the SceneGraph can be found in the RT directory. The SceneGraph uses a **require** framework which ensures that all required scripts are load into the application. You can find more details about this in the tutorials section of this document, covering the creation of a custom SceneGraphNode.

2.2. JavaScript and KL

Types can provide extra functionality both in JavaScript as well as Fabric Engine's kernel language (**KL**). The type below implements an **Address**, and implements both a JavaScript file as well as a KL file providing the additional functions, called the **bindings**. The content of the JavaScript file looks like this: (the first array parameter is the list of requirements. this could include other types, for example.)

```
FABRIC.define([], function() {

  // Constructor:
  FABRIC.RT.Address = function(options) {
    if(!options) options = {};
    this.street = options.street != undefined ? options.street : '';
    this.number = options.number != undefined ? options.number : 1;
    this.city = options.city != undefined ? options.city : '';
    this.zip = options.zip != undefined ? options.zip : 1000;
    this.country = options.country != undefined ? options.country : '';
  };

  // Prototype, providing additional methods
  FABRIC.RT.Address.prototype = {
    getPrintable: function() {
      return this.street + ' ' + this.number + ' in ' + this.zip + ' ' + this.city + ', '
 + this.country;
    }
  };

  // Append this type to be loaded once the Fabric context exists
  FABRIC.appendOnCreateContextCallback(function(context) {
    context.RegisteredTypesManager.registerType('Address', {
      members: {
        street: 'String', number: 'Integer', city: 'String', zip: 'Integer', country: 'String'
      },
      constructor: FABRIC.RT.Address,
      klBindings: {
        filename: 'Address.kl',
        sourceCode: FABRIC.loadResourceURL('RT/Address.kl')
      }
```

```
    });
  });

  return FABRIC.RT.Address;
});
```

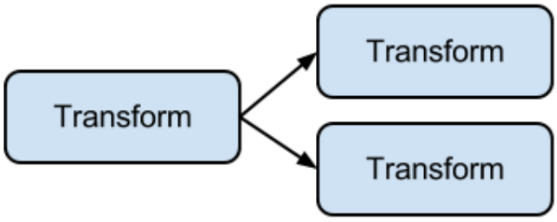
The KL file mentioned in the appendOnCreateContextCallback above, **RT/Address.kl** could look like this:

```
function String Address.getPrintable() {
  return this.street + ' ' + this.number + ' in ' + this.zip + ' ' + this.city + ', '
 + this.country;
}
```

This will make the **getPrintable** method available both in JavaScript as well as in KL. As you can see, you can specify any type of data. You can of course include custom types as members of other custom types, and therefore build very complex structures, which then can be used for computation inside the dependency graph or MapReduce.

Chapter 3. Kinematics Guide

3.1. Basic Transform



The SceneGraph's basic transform can represent a global or a hierarchical transform. It can be contructed like this:

```
var globalTransform = scene.constructNode('Transform', {
  hierarchical: false
});
var hierarchicalTransform = scene.constructNode('Transform', {
  hierarchical: true,
  parentTransformNode: globalTransform
});
```

When contructed in global mode (hierarchical: false), the transform simply constructs global matrix based on the localXfo value. When contructed in hierarchical mode, a parent Transform node must also be provided. The transform node builds its global Xfo by multiplying the parent's global Xfo by its own local Xfo to get a new global Xfo. Hierarchies of transforms can be connected together to drive complex hierarchical animation.

When creating the transform node, you can specify the global or the local transform by providing a **FABRIC.RT.Xfo**:

```
var globalTransform = scene.constructNode('Transform', {
  hierarchical: false,
  globalXfo: new FABRIC.RT.Xfo({
    tr: new FABRIC.RT.Vec3(1.0, 2.0, 3.0)
  })
});
```

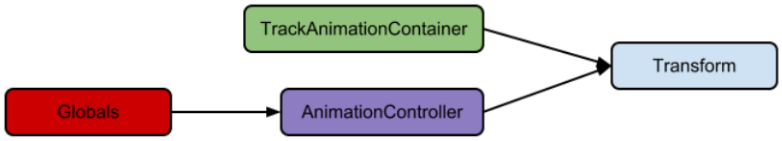
3.2. Aimed Transform

For the typical *lookat* behaviour of a transform, for example used for a camera with an lookat position, or a spot light with a target position, the SceneGraph provides a specialized transform node called the **AimTransform**. It adds an additional option to the factory function, which allows you to specify the target position.

```
var aimTransform = scene.constructNode('AimTransform', {
  globalXfo: new FABRIC.RT.Xfo({
      tr: new FABRIC.RT.Vec3(1.0, 2.0, 3.0)
  }),
  target: new FABRIC.RT.Vec3(10.0, 0.0, 0.0)
});
```

The upvector direction used for the AimTransform is always the positive Y axis.

Chapter 4. Animation Guide



The animation pipeline in Fabric Engine's SceneGraph is inspired by the animation systems commonly found in game engines. The SceneGraphNodes in the animation pipeline are:

- **AnimationController**
- **TrackAnimationContainer**
- **CharacterAnimationContainer**

The evaluation of the animation is done in KL operators on the target nodes. In the image above the animation is driving a transform node, but any member on any kind of node can be driven by animation in Fabric Engine's SceneGraph.

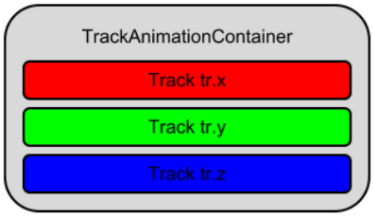
4.1. The Animation Controller node

An Animation Controller is responsible for computing a time value from given inputs. The Animation Controller takes in a time paramter and uses it to compute a new time value. For example, an Animaiton Controller might take in global time and compute a, loop or ping-pong time value which is then used to drive an animation evaluation. A single controller can be used by many evaluators. The default Animation Controller node is bound to the globals SceneGraph node storing the global scene time.

```
var controller = scene.constructNode('AnimationController', {
  timeRange: new FABRIC.RT.Vec2(0.0, 10.0), // ten seconds
  outOfRange: 1 // looping
});
```

4.2. The TrackAnimationContainer

Both the **TrackAnimationContainer** as well as the **CharacterAnimationContainer** inherit from the **AnimationContainer**. The AnimationContainer SceneGraph node represents a container for any kind of animation. The TrackAnimationContainer uses a sliced dependency graph node to store many animation tracks in a single node. All tracks on the TrackAnimationContainer have to contain the same type of keys, which is why you can't construct a TrackAnimationContainer as is, you need to instantiate inherited nodes of it, such as the **LinearTrackAnimationContainer** containing linear keys or **BezierTrackAnimationContainer** containing bezier keys.



The TrackAnimationContainer represents a single animation of multiple tracks. The benefit of the TrackAnimationContainer is that tracks are stores in slices, and processing on the tracks can happen in a multi-threaded fashion.

```
var linearkey = function(time, value){ return new FABRIC.RT.LinearKeyframe(time, value); };
```

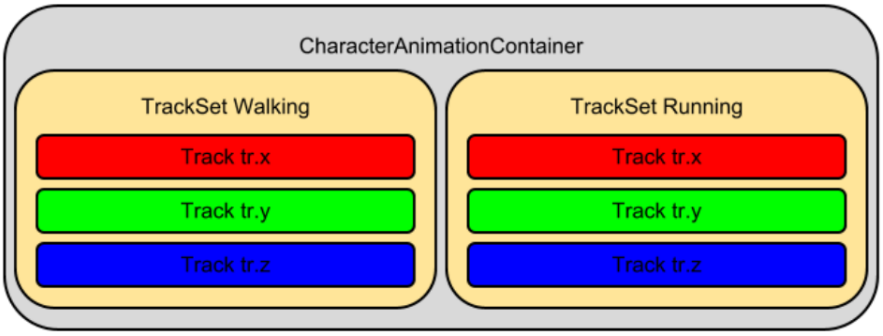
```
var trackContainer = scene.constructNode('LinearTrackAnimationContainer', {});

trackContainer.addTrack(new FABRIC.RT.LinearKeyframeTrack('tr.x', FABRIC.RT.rgb(1, 0, 0), [
  linearkey(0, 0), linearkey(50, 0), linearkey(75, 50), linearkey(100, 0)
]));
trackContainer.addTrack(new FABRIC.RT.LinearKeyframeTrack('tr.y', FABRIC.RT.rgb(0, 1, 0), [
  linearkey(0, 0), linearkey(50, 10), linearkey(75, 00), linearkey(100, 0)
]));
trackContainer.addTrack(new FABRIC.RT.LinearKeyframeTrack('tr.z', FABRIC.RT.rgb(0, 0, 1), [
  linearkey(0, 0), linearkey(50, 30), linearkey(75, 00), linearkey(100, 30)
]));
```

The code above create a small helper function for creating a linear key frame, then the track animation container is created. Once the container exists tracks can be pushed to it, providing the name of the track, a color for the UI as well as an array of keys for the track.

4.3. The CharacterAnimationContainer

The **CharacterAnimationContainer** on the other hand is quite different from the TrackAnimationContainer. It stores a complete animation containing several tracks per slice, allowing to store a large number of complete animations on a single node. This container can be understood as a library of animations. Each animation is stored as a **TrackSet**, which is essentially an array of tracks as well as a name for the TrackSet.



The CharacterAnimationContainer is used for the character pipeline, and will be discussed in more detail in the characters related section of this document. Essentially though, this is how you create a CharacterAnimationContainer:

```
var linearkey = function(time, value){ return new FABRIC.RT.LinearKeyframe(time, value); };

var characterContainer = scene.constructNode('LinearCharacterAnimationContainer', {});

var walking = characterContainer.newTrackSet('walking');
walking.tracks.push(new FABRIC.RT.LinearKeyframeTrack('tr.x', FABRIC.RT.rgb(1, 0, 0), [
  linearkey(0, 0), linearkey(50, 0), linearkey(75, 50), linearkey(100, 0)
]));
walking.tracks.push(new FABRIC.RT.LinearKeyframeTrack('tr.y', FABRIC.RT.rgb(0, 1, 0), [
  linearkey(0, 0), linearkey(50, 10), linearkey(75, 00), linearkey(100, 0)
]));
walking.tracks.push(new FABRIC.RT.LinearKeyframeTrack('tr.z', FABRIC.RT.rgb(0, 0, 1), [
  linearkey(0, 0), linearkey(50, 30), linearkey(75, 00), linearkey(100, 30)
]));

characterContainer.addTrackSet(walking, [0,1,2]);
```

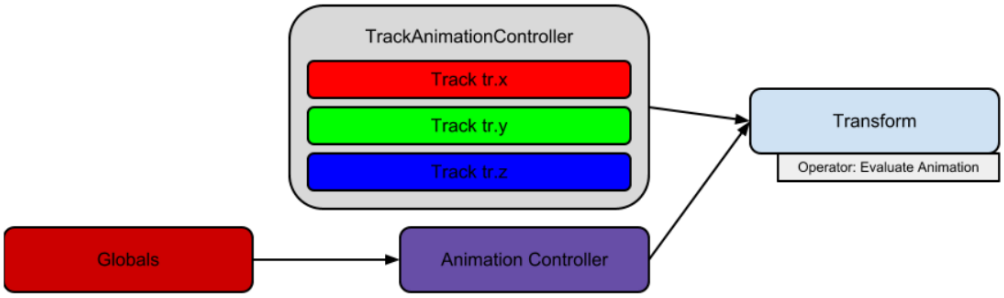
4.4. Binding animation to target nodes

To use the animation on a target node, such as a transform node, for example, you can call the **bindNodeMembersToTracks** method of the animation container. You need to specify the target binding which includes the name of the

member on the target node as well as the tracks that need to be bound to it. This you can drive complex types, as as a **Vec3** for example, you can bind multiple tracks to the same member. In the case of a Vec3 the first track will go to the x component, the second track to the y component and so on.

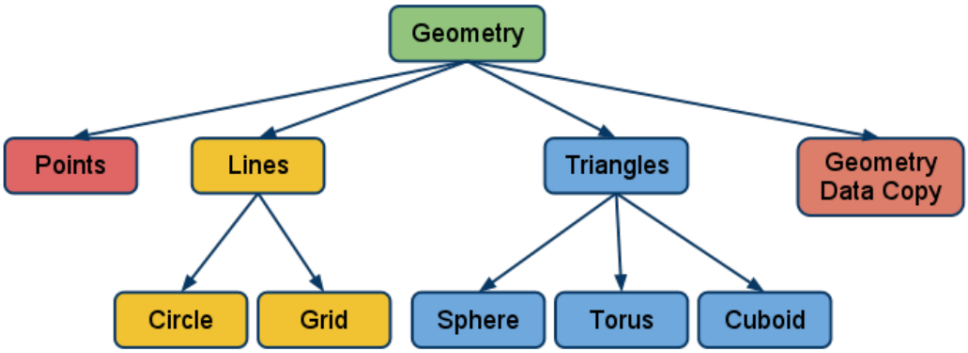
```
trackContainer.bindNodeMembersToTracks(transform, {
  'globalXfo.tr': [0, 1, 2] // representing the tracks tr.x, tr.y and tr.z
}, controller);
```

Binding the animation on the target node will create an operator on the target's dependency graph node which evaluates the animation provided in the animation container and writes to the target node's member data. The operator also binds in the animation controller which was used during the construction of the animation container. If you don't provide a controller to the bindNodeMembersToTracks method the generated operator will simply bind to the global time. So in the case discussed in the code snippets in the animation guide, the resulting graph will look like this:



Chapter 5. Geometry Guide

The geometry hierarchy in Fabric Engine's SceneGraph follows a classical inheritance model. Each node in the hierarchy adds functionality the the constructed SceneGraph node. Geometry can be created in several ways:



1. A base geometry type can be constructed, and its geometry data can be populated from JavaScript:

```
var geometryNode = scene.pub.constructNode('Triangles');
geometryNode.loadGeometryData({
  positions: [FABRIC.RT.vec3(0, 0, 0), FABRIC.RT.vec3(0, 0, 1.0), FABRIC.RT.vec3(1.0, 0, 0)],
  indices: [0, 1, 2]
});
```

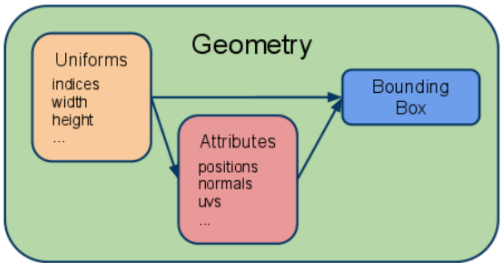
2. Geometry can be constructed by loading an external resource file, such as an OBJ file, for example:

```
scene.importAssetFile('Models/cow.obj',{ splitMaterials: true } );
```

3. A primitive can be constructed. The primitive constructor assigns geometry generation operators which create all of the vertices and other data such as normals based on options on the primitive.

```
var primitiveNode = scene.constructNode('Circle', { radius: 7 });
```

5.1. Anatomy of a Geometry Node



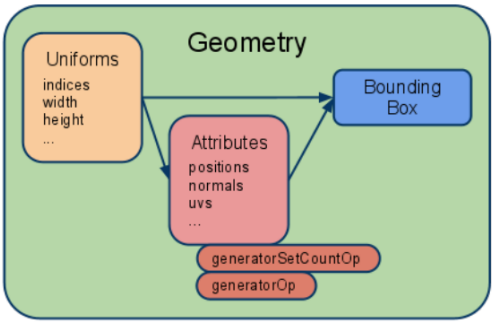
A geometry SceneGraphNode contains several dependency graph nodes. Fabric Engine's slicing scheme is used to represent the data, and to allow high performance operations such as deformation on the geometry using multi-threading.

The **Uniforms** node stores single values that are associated with the geometry, but don't vary per component. Primitives assign uniform values that are often exposed from the scene graph node as modifiable parameters. Triangles and Lines also store an array of Integers on the Uniforms node called **indexList** which defines the connectivity of points for lines and triangles.

The **Attributes** node stores all the ‘per-component’ information. Attributes can easily be added and removed from geometry by simply adding and removing members from this node. The Attributes node is sliced, so the number of slices matches the number of vertices, for example.

The **BoundingBox** node is generated using the attributes, and is used to accelerate interaction with the geometry such as raycasting.

5.2. Geometry Generation



Note: this are will be re-structured slightly once we have support for node nesting.

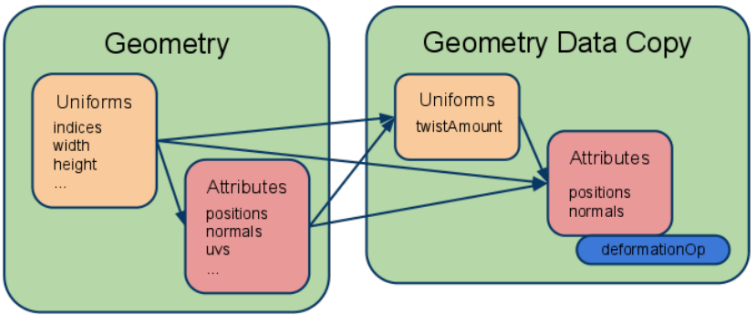
The purpose of a generator operator is to generate the vertex information that is stored in the attributes node, and the indexList (for Triangles and Lines), which is stored in the 'Uniforms' node. First we calculate the number of vertices in the **setCount** operator, and then populate the attributes node in the second operator. The second operator also writes the indexList which is stored on the Uniforms node.

5.3. Geometry Services

A key service that the the Geometry node provides is integration with the rendering pipeline. The design of Fabric Engine's SceneGraph is based heavily on OpenGL shading language GLSL, and all rendering in the OpenGL viewport is performed via shaders.

The geometry node constructs an event handler and binds operators to it for loading all the various vertex attributes into the GPU and storing the buffer ids. This means that you can add a new vertex attributes to a geometry, then use that data in a shader and the geometry node will take care of loading the data so it is ready for the shader. This is discussed in more detail in the **Drawing Pipeling Guide** section of this document.

5.4. Geometry Data Copies



Geometry data copies provide a way to split the definition of a rendered geometry into multiple sections. A Geometry data copy is used to extend a base geometry, usually by adding deformation operators. The base geometry represents the original undeformed geometry, and the data copy is used to apply changes. The Geometry data does not contain a complete description of gometry, but only the the data that is a modification of the base geometry. During rendering,

uniforms and attributes are loaded from the base geometry and the data copy where any attributes in the data copy verride the attributes in the base geometry.

Since the Attributes node is sliced, deformation can be performed using multi-threading.

```
var geometryCopyNode = scene.constructNode('GeometryDataCopy', {baseGeometryNode:
 geometryNode} );
geometryCopyNode.addVertexAttributeValue('positions', 'Vec3', { genVBO:true, dynamic:true } );
```

Note: The *PerPointDeformation.html* sample application contains an example of this.

Chapter 6. Parsers Guide

Fabric Engine's SceneGraph comes with several parsers for importing external resource files. Since the loading of external data happens asynchronously, you will need to provide callback functions to the parsers, which will executed once the content is loaded, parsed and ready for use. Parsers are automatically invoked and chosen based on the file extension of the resource to load.

6.1. The OBJ Parser

The OBJ file format can store polygonal meshes with UV coordinates as well as basic material settings. You can use the OBJ parser like this:

```
scene.importAssetFile('Models/cow.obj', {splitMaterials: false}, function(assetNodes) {
  for(var name in assetNodes) {
    console.log(assetNodes[name]);
  }
});
```

The last argument to the **importAssetFile** method is the callback function to execute once the parsed result is ready for consumption. The OBJ parser returns a dictionary of triangle geometry nodes representing all of the geometries which are part of the OBJ file. If the **splitMaterials** option is set to true, the Obj parser will splith the geometries into multiple geometry sets according to the material assignments in the file. Each returned geometry can then be assigned separate materials for rendering.

Note: The *ModelViewer.html* sample application contains an example of the OBJ parser in use.

6.2. The Collada Parser

Collada's DAE file format can store complex types of data, including polygonal meshes, transform hierarchies, point clouds as well as other complex 3D data structures. The SceneGraph's collada parser currently supports:

- Polygonal Meshes including UV coordinates and skinning weights, imported as Triangles nodes
- Character Hierarchies used for skinning, imported as CharacterRig nodes
- FCurve animation on hierarchies, imported as CharacterAnimationContainer nodes
- Cameras including focal length and aperture settings, imported as Camera nodes

The Collada parser can be invoked like this:

```
scene.importAssetFile('Models/character.dae', {
    constructRigFromHierarchy: true
}, function(assetNodes) {
  for(var name in assetNodes) {
    console.log(assetNodes[name]);
  }
});
```

The last argument of the parser's invocation is a callback function to be called once the collada data is parsed and ready for consumption. In the example above we simply log the data to the console. For further details on the character related nodes please see the **Character Guide** section of this document.

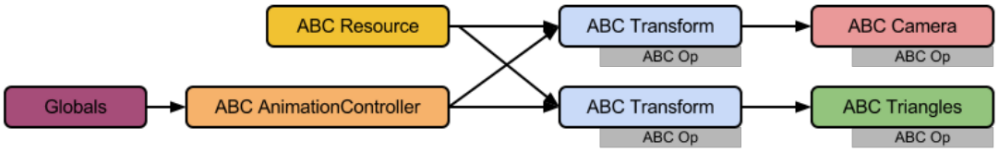
Note: The *CharacterSkeleton.html* sample application contains an example of the Collada parser in use.

6.3. The Alembic Parser

Alembic ABC file format can store several kinds of data. The SceneGraph's Alembic intergration currently supports:

- Polygonal Meshes including UV coordinates and normals, imported as Triangle nodes
- Curve lists including width and color values, imported as Lines nodes
- Cameras with proper focal length and aperture settings, imported as Camera nodes
- Point clouds including full transform support, color values, imported as Points nodes

Since Alembic stores animation as discrete samples the parser doesn't load all of the data to memory, but rather constructs operators which re-evaluate based on time changes. This way animation can be loaded into Fabric Engine and synchronized with the SceneGraph's global time or a different AnimationController.



The Alembic resource node holds what is called a **AlembicHandle** to the original resource file, which is forwarded through the dependency graph and used by all of the ABC operators to pull the relevant data out of the Alembic file. Per se there is no parsing happening in JavaScript, so once the resource is loaded, the **loadSuccess** event fires. The sample code below shows how to load data from an Alembic resource. For more details on event handling please refer to the **Event Guide** section of this document.

```
var alembicLoadNode = scene.constructNode('AlembicLoadNode', {
  url: 'Models/cow.abc'
});
alembicLoadNode.addEventListener('loadSuccess', function(){
  var assetNodes = alembicLoadNode.getParsedNodes();
  for(var name in assetNodes) {
    console.log(assetNodes[name]);
  }
});
```

Note: The *Alembic/Primitives.html* sample application contains an example of the Alembic parser in use.

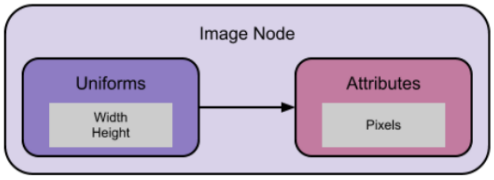
Chapter 7. Characters Guide

This is do be done for Phil!

Chapter 8. Images and Video Guide

The SceneGraph supports reading and using external image resource files as well as external video streams. Typically images are never loaded onto dependency graph nodes or stored in the main memory, they are rather pushed directly to the GPU. When drawing textures there is no need to keep the image in the RAM, however if you want to modify the image using the CPU the images has to be stored in the dependency graph. Fabric Engine supports 2D images as well as 3D images.

8.1. Anatomy of an Image node



Similar to the Geometry node, an image node is split into two dependency graph nodes. The **Uniforms** node stores all of the non-per-pixel data, such as the width and height of the image. These values are typically exposed to the SceneGraph node as getter function. The **Attributes** node contains the per pixel data, however only if the Attributes dependency node is required at all. As mentioned above the per pixel data is normally directly pushed to the GPU.

8.2. Loading 2D Images

Currently the SceneGraph supports loading 2d images of the types **JPG**, **PNG**, **BMP**, **TIF**, **EXR**, and **HDR**. The last two are stores as floating point images, while the other formats are stored as byte images. 2d images are always stored as RGBA, even if the original image data doesn't contain an alpha channel.

```
var image2D = scene.constructNode('Image2D', {
  url: 'Resources/tomatoes_960_640.png',
  createDgNodes: false
});
```

If you want the image to be loaded into the RAM and accessible in the dependency graph, you need to set the **createDgNodes** option to true. You can then attach operators to the Attributes node to perform per pixel calculations in a multi-threaded fashion.

Note: The *BackgroundTexture.html* sample application contains a simple example of an image node in use.

8.3. Loading 3D Images

Currently the SceneGraph only supports loading 3d images of the type **NRRD**. 3d images are always stored as USHORT with RGBA channels.

```
var image3D = scene.constructNode('Image3D', {
  url: 'Resources/threed_texture.nrrd',
  createDgNodes: false
});
```

As with 2d iamges, if you want the image to be loaded into the RAM and accessible in the dependency graph, you need to set the **createDgNodes** option to true.

Note: The *MedicalImaging.html* use-case application contains an example of a 3d image node in use.

8.4. Loading Video

Video nodes are pretty much the same as Image2D nodes, except that they have operators attached which pull frames out of the external video file resource. For that, the current frame is stored on the Attributes dependency graph node and is constantly pushed to the GPU if the time of the video changes. Fabric Engine utilizes the **FFMPEG** library to offer support for a wide variety of video formats. Aside from other formats, the SceneGraph's video node supports: **AVI**, **MOV** and **MP4**.

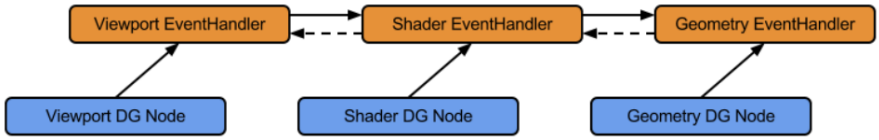
```
var videoNode = scene.constructNode('Video', {
  url: 'Resources/bee_960.mov',
  loop: true
});
```

Note: The *Video.html* sample application contains a simple example of a video node in use.

Chapter 9. Drawing Pipeline Guide

Fabric Engine's SceneGraph's drawing pipeline is very flexible and customizable. Drawing is performed with as few operations as possible providing high performance rendering. Generally is is important to understand that drawing invokes the evaluation of the dependency graph.

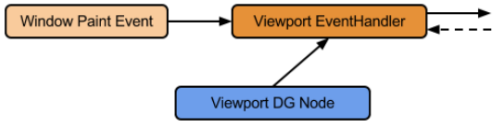
9.1. Predescend and Postdescend



The drawing pipline is implemented using eventhandlers. The eventhandler tree for drawing listens to the paint event of the canvas resp. the windows. The drawing pipeline evaluates in two phases: the **Predescend** and **Postdescent**. Predescend happens when the drawing pipline evaluates each event handler from left to right, resp. from the event down to the last eventhandler. Postdescend happens once the descend is done, traveling through the event graph back up to the event. Operators for drawing can be applied on either the **preDescendBindings** or the **postDescendBindings.** When each eventhandler evaluates it pulls on the connected dependency graph nodes. If the dependency graph node itself has dependencies and operators, the execution of the dependencies and operatos will be fired.

9.2. Windows

Each window in Fabric defines a dependency graph node and an Event. The node contains data about the window such as its width and height, and the event is fired when painting of the window is required. The width and height values are driven by the Fabric plugin, and are updated when the window is resized.



The Window Event node is an event that is fired whenever the window needs painting. This may occur because the window was resized, revealed, or invalidated in any way. It is also possible to manually trigger a redraw from JavaScript. Redrawing of the window during animation is effected by modifying graph variables such as time, and then manually triggering a redraw. The JavaScript mechanism to fire a redraw manually is accessible through the **Viewport** node.

```
var viewport = scene.constructNode('Viewport', {
  windowElement: document.getElementById('FabricContainer') // provide the HTML element for the
 viewport
});
// fire a redraw manually
viewport.redraw();
```

By attaching event handlers to the Window Paint Event node, we can coordinate the drawing of the window using OpenGL. The OpenGL context is set up and bound prior to the paint event being fired. For more information on Events, and Event Handlers, please consult the Core Programming Guide.

9.3. Setting up the Viewport in OpenGL

When the window redraw event is fired, the subgraph is traversed, first visiting the Viewport event handler which has an operator attached called **viewPortBeginRender**. When viewPortBeginRender is executed, the initial OpenGL parameters are set.

```
operator viewPortBeginRender(io Integer width, io Integer height, io Color backgroundColor) {
  glCullFace(GL_BACK);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);   glViewport(0, 0, width, height);
  glClearColor(backgroundColor.r,
  backgroundColor.g, backgroundColor.b, backgroundColor.a);
  glClear(GL_COLOR_BUFFER_BIT |
  GL_DEPTH_BUFFER_BIT);
}
```

The camera event handler is then visited, where the camera projection values are computed in the operator **updateCameraProjection**.

9.4. Shaders

The Fabric SceneGraph rendering system breaks shaders into 2 components, **shaders** and **materials**. The shader event handler loads the shader source into the OpenGL driver and compiles the shader program. The specification of the shader also includes some meta-data such as the buffers required by the shader, and any constant values that must be defined.

Note: To inspect the drawing pipeline you can use the SceneGraph's debugger. Just open the *ModelViewer.html* sample application, for example, open the JavaScript console, and type:

```
FABRIC.displayDebugger();
```

9.5. GLSL Uniforms and Attributes

GLSL shaders define a set of parameters which must be filled out before the geometry can be drawn on screen. These 2 sets of requirements are stored in 2 arrays called attributeValues, and uniformValues. These arrays are accessible by all child event handlers who can determine what data is required by the shader.

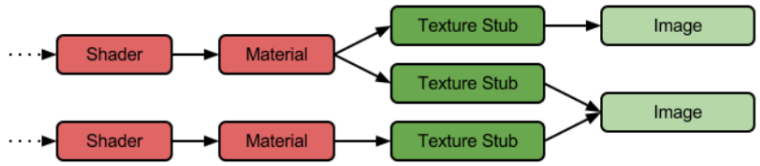
Note: the design of the event handler graph means that 2 different nodes can define the same data under the same 'scope name', and the order of visitation defines which scope is valid. This enables a single geometry to be rendered multiple times using different shaders.

9.6. Materials

Materials define parameters for the shader program and is used to attach image loaders. The Material node in Fabric Engine is generated using meta data associated with the shader. XML files are loaded which contain both the GLSL shader code, and meta data that instructs the rendering system how to construct material node instances.

Materials provide data to the shader such as shading parameters, and also provide a branch point in the event graph where textures can be loaded and bound prior to traversal continuing to the Instance node.

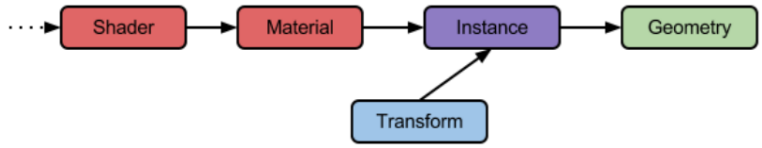
9.7. Textures



Materials which utilize textures must specify which **textureUnit** to assign each texture. This process is automated using the material system in the SceneGraph. For each textureUnit, a **stub** node is generated. This stub node binds in the texture unit that will be assigned to the texture. This allows multiple textures to be used on the same material, and textures to be shared amongst materials.

9.8. Instances

The **instance** node represents a drawn piece of geometry. To be able to draw geometry on screen, several things bust have happened prior to the draw call.

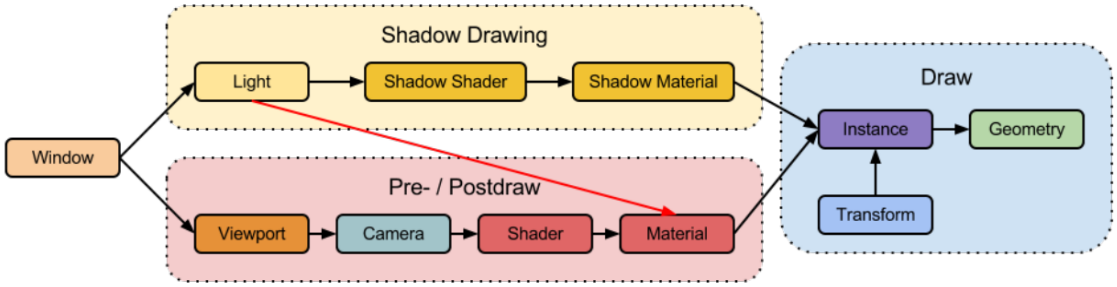


Geometry rendering happens in the following order:

1. The shader program is loaded
2. The the material node is traversed and shader constants set
3. From the material traversal may continue into the texture stubs and on into the texture nodes where the textures are loaded.
4. From the material node, traversal continues down to the instance node, where the matrix for the draw is loaded.
5. From the instance node, traversal continues down to the geometry node where the geometry buffers are loaded and shader uniforms may also be set.
6. The Geometry is a leaf node, and after the preDescendBindings have been evaluated the postDescendBindings follow immediately, and the draw call is executed.

9.9. Shadows

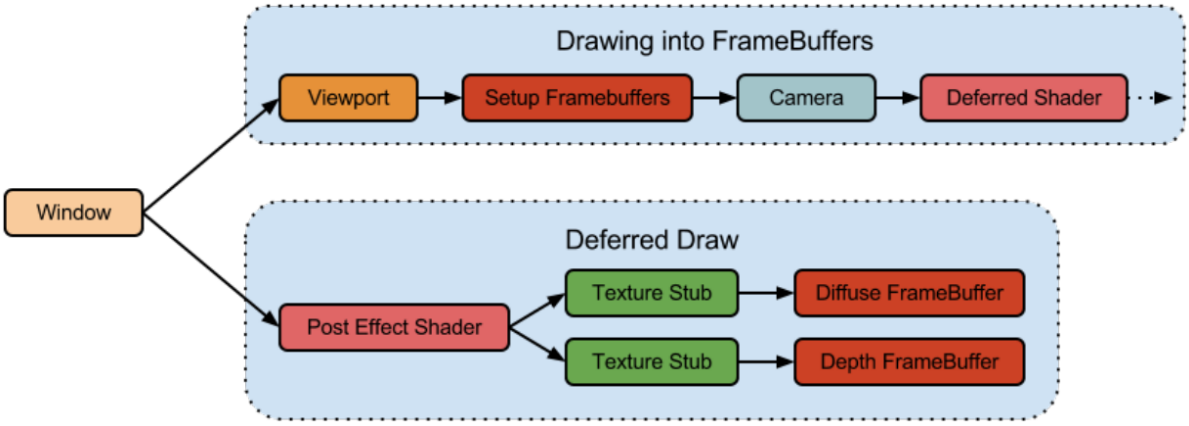
Shadow maps are rendered in a pre-draw stage prior to the camera render pass. For each light, we render the scene from the point of view of the light, and store the depth information in a depth buffer. The depth buffer ID is stored in the light node which is also bound to the material. This gives the material access to the lights shadow buffers, enabling the correct shadowing of geometry.



The red arrow in the diagram represents a binding from the light to the material. This binding is what gives the material access to the light's shadow buffer.

Chapter 10. Deferred Rendering Guide

Deferred rendering describes the process of drawing the viewport's content to several framebuffers, and then doing additional passes of rendering utilizing these framebuffers. Deferred rendering is tremendously different from standard rendering, since all of the GLSL shaders have to store their resulting pixel values into framebuffers. So shaders have to be specifically written for deferred rendering.



Deferred rendering allows to perform 2D post effects on the GPU, or additional 3D rendering as a post effect.

Note: The *ToonRendering.html* sample applications contains an example of using the deferred renderer.

10.1. Setting up the Deferred Renderer

The SceneGraph provides a utility node called the **BasicDeferredRenderer**.

```
var renderer = scene.constructNode('BasicDeferredRenderer', {
  addDepth: false, // don't add a depth framebuffer
  colorBuffers: [{name: 'diffuseA, nbChannels: 4}, {name: 'diffuseB', nbChannels: 4}]
});
```

Once the renderer is setup, you can setup materials through it. This will ensure that the shader and material eventhandlers are connected up correctly within the drawing pipeline. You can create **prePassMaterials** (for the initial draw into framebuffers) or **postPassMaterials** (for the deferred draw utilizing the framebuffers).

```
var drawMaterial = renderer.addPrePassMaterial('MyDeferredShader', {});
var compositingMaterial = renderer.addPostPassMaterial('MyCompositingShader', {});
```

10.2. PrePass Deferred Shaders

Shaders for a deferred rendering pipline have to write their results in the fragment data in GLSL. This will fill the framebuffers. You need to setup the framebuffers accordingly when constructing the deferred renderer node. Prepass deferred shaders replace the standard shaders from a non-deferred drawing pipeline. Here's an example of a small fragment shader that writes red to the first and blue to the second framebuffer.

```
void main(){
  gl_FragData[0] = vec4(1.0,0.0,0.0,1.0);
  gl_FragData[1] = vec4(0.0,0.0,1.0,1.0);
}
```

10.3. PostPass Deferred Shaders

PostPass deferred shaders perform the **compositing** of the framebuffers to the final displayed image. Typically postpass shaders perform per pixel compositing, but they can access all pixels of the framebuffers, since the framebuffers are stored as textures on the GPU. Here's an example of a fragment shader which combines both diffuse color from the deferred renderer above:

```
uniform sampler2D u_samplerDiffuseA;
uniform sampler2D u_samplerDiffuseB;

void main(){
  vec2 windowCoord = gl_TexCoord[0].st*0.5+0.5;
  gl_FragColor = texture2D( u_samplerDiffuseA, windowCoord ) + texture2D( u_samplerDiffuseB,
 windowCoord );
}
```

This will result in a violet rendering of the 3D scene, combined from two different draw results.

Note: Please also see the *DeferredRendering.html* sample application for an example of a more advanced scenario.

Chapter 11. SceneGraph IO Guide

Waiting for Jerome's input...!

Chapter 12. Events Guide

The SceneGraph provides a system for automatically firing callbacks when certain things happen. SceneGraph events are a very powerful mechanism to automate functionality, especially since in web applications things can happen asynchronously.

12.1. Setting up a node for event handling

To enable events on a SceneGraph node the **addEventHandlingFunctions** method has to be called in the constructor of the node.

```
scene.addEventHandlingFunctions(myNodePrivateInterface);
```

Once that's done the node can fire events and receive event callbacks. To attach a callback to the node, other nodes resp. the scene script can call the **addEventListener** method.

```
var myCallback = function(evt) {
  console.log(evt);
  return 'remove';
};
myNodePublicInterface.addEventListener('myEvent', myCallback);
```

If the event callback function returns *'remove'* the callback will be removed from the event listener stack, and won't fire again if the element fires another time. If you don't return *'remove'* the callback will stay attached.

You can also remove the event callback by calling the **removeEventListener** method, which allows to have functions listens to event dynamically. So to say you can attach functions and de-attach them again.

```
myNodePublicInterface.removeEventListener('myEvent', myCallback);
```

To trigger the event programatically you can call the **fireEvent** method on the node.

```
myNodePublicInterface.fireEvent('myEvent', { data: 'myData' });
```

This fires all attached event callbacks, and provides the event data to each of the callback functions. This mechasnism is used heavily throughout the SceneGraph.

12.2. loadSuccess event

The **ResourceLoadNode**, covered in the **SceneGraph IO Guide** in this document, uses events to dispatch the call of event listeners once a resource file is loaded. This mechanism is uses in the SceneGraph parsers, but you can also use it for custom scenarios like this:

```
var resourceLoadNode = scene.constructNode('ResourceLoad', {
  url: 'myBinaryFile.bin'
});
resourceLoadNode.addEventListener('loadSuccess', function(evt){
  // attach operators to the resourceLoadNode for parsing...
  return 'remove';
});
```

This allows you to automatically continue with the setup of the application once the resource file is downloaded and ready.

Chapter 13. Selection Manager Guide

The SceneGraph comes with a manager for selection, called the **SelectionManager**. This manager is the basic implementation, and simply keeps track of a collection of selected nodes. It provides the **selectionChanged** event for automating UI changes, for example, based on the selection.

13.1. Basic Selection Management

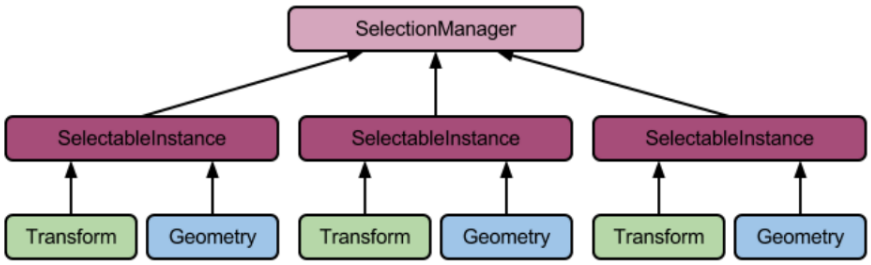
The code snippet below sets up the manager, attaches a callback to it and selects as well as deselects a single node. This code is the basis for more complex selection management.

```
// setup manager and attach event callback
var selectionMgr = scene.constructManager('SelectionManager');
selectionMgr.addEventListener('selectionChangesd',function(evt) {
  console.log(evt.selection);
});

// select a node and deselect it again
selectionMgr.addToSelection(myNode);
selectionMgr.removeFromSelection(myNode);
```

13.2. Viewport Selection Manager

The **ViewportSelectionManager** works by providing a new **Instance** node, called the **SelectableInstance**. It contains a selection state as well as an additional material, that indicates the selection during drawing.



When a SelectableInstance node is hovered, the material on the node changes to the *highlighted* material. Once the node is selected the material changes to the *selected* material.

```
// create the materials
var normalMaterial = scene.constructNode('FlatMaterial', { color: FABRIC.RT.rgb(0.0, 0.0, 1.0,
 1) });
var highlightMaterial = scene.constructNode('FlatMaterial', { color: FABRIC.RT.rgb(0.7, 0.7, 0.7,
 1) });
var selectedMaterial = scene.constructNode('FlatMaterial', { color: FABRIC.RT.rgb(1.0, 0.0, 0.0,
 1) });

// create a selectable instance
var myInstance = scene.constructNode('SelectableInstance', {
  transformNode: myTransform,
  geometryNode: myGeometry,
  materialNode: normalMaterial,
  highlightMaterial: highlightMaterial,
  selectMaterial: selectedMaterial
});

// setup manager and attach event callback
var selectionMgr = scene.constructManager('ViewportSelectionManager');
```

```
selectionMgr.addEventListener('selectionChangesd',function(evt) {
  console.log(evt.selection);
});

// select a node and deselect it again
selectionMgr.addToSelection(myInstance);
selectionMgr.removeFromSelection(myInstance);
```

Note: The Selection.html sample application contains an example of how to use selection in the SceneGraph.

13.3. Selection Manipulation Manager

The **SelectionManipulationManager** provides an additional mechanism for selection based manipulation. For further details on the way manipulation works in the SceneGraph please refer to the **Manipulation Guide** in this document. The SelectionManipulationManager itself is not taking care of the selection management, therefore you need to specify the SelectionManager during its construction. The SelectionManipulationManager just takes care of setting up a group manipulation, that toggles on and off based on selection changes as well as centers the manipulator in the center of the selected objects.

```
var manipulationMgr = scene.constructManager('SelectionManipulationManager', {
  selectionManager: selectionMgr,
  manipulators: undefined // here you can specify an array of manipulators to manage
});
```

By providing a list of manipulators to the *manipulators* option, you can define the kind of manipulations you want the manager to take care of. If you don't specify anything, the manipulation will default to to a screen space translation.

Chapter 14. Manipulation Guide

The SceneGraph provides a solid framework for manipulation. While a series of transform manipulators are provided, you can build your own manipulators. This document includes a **Tutorial for building a custom manipulator**.

Manipulation in the SceneGraph works by modifying the dependency graph's data in JavaScript. For this, several steps of execution are required:

1. On MouseDown: Gather all of the current relevant values (snapshot) and attach event callbacks for MouseMove
2. On MouseMove: Compare current values with the snapshot and change the graph accordingly
3. On MouseUp: Remove the MouseMove event callbacks.

14.1. Setting up a Manipulator

Manipulators require **RayCasting** to be enable in the viewport, since it needs to determine which objects are hit below the mouse's position. To enable RayCasting, provide the following options when constructing the viewport:

```
var viewport = scene.constructNode('Viewport', {
  windowElement: document.getElementById('FabricContainer'),
  enableRaycasting: true,
  rayIntersectionThreshold: 0.8
});
```

Manipulators operate on Instance nodes, so you first have to create an instance node to be able to attach a manipulator. Manipulators can operate on any member on any dependency graph node, but since typically manipulation in 3D affects the transforms of objects, all of the provided manipulators in the SceneGraph operate on the **globalXfo** member of the transform node.

```
// create an instance out of geometry, transform and material
var instance = scene.constructNode('Instance', {
  geometryNode: myGeometry,
  transformNode: myTransform,
  materialNode: myMaterial
});

// create a manipulator for the instance node
var manipulator = scene.constructNode('3AxisTranslationManipulator', {
  targetNode: instance,
  size: 10,
  linearTranslationManipulators: true,
  planarTranslationManipulators: true,
  screenTranslationManipulators: true
});
```

For a list of all of the manipulators available and their options please refer to the **SceneGraph Node Reference** in this document.

Note: The Manipulators.html sample application contains an example of how to use manipulation in the SceneGraph.

Chapter 15. Undo / Redo Guide

Undo and Redo functionality in the SceneGraph can be achieved by constructing the **UndoManager** and pushing **Transactions** to it. A transaction is a dictionary containing callbacks for **onClose**, **onUndo** and **onRedo**. The SceneGraph's manipulation system provides these kinds of transactions already, but you can implement your own.

To undo or redo a transaction, simply call the on the **undo** and **redo** methods of the of the UndoManager.

```
// create an undo manager
var undoMgr = scene.constructManager('UndoManager');

// create a value to be changed
var value = 10;
var prevValue = value;
var newValue;

// push a transaction to it
undoMgr.addTransaction({
  onClose: function() {
    newValue = value;
  },
  onUndo: function() {
    value = prevValue;
  },
  onRedo: function() {
    value = newValue;
  }
});

// change the value
value = 20;

// close the transaction
undoMgr.closeTransaction();

// undo the change
undoMgr.undo();
```

Note: The Undo.html sample application contains an example of how to use Undo and Redo in the SceneGraph.

15.1. Using Undo and Redo with Manipulation

The manipulation system already supports undo. Since you can have multiple UndoManagers in an application, you can optionally provide the UndoManager to the construction options of the manipulators, but even if you didn't specify it the first created undomanager will be used for manipulation.

```
// create an undo manager
var undoMgr = scene.constructManager('UndoManager');

// create a manipulator for the instance node
var manipulator = scene.constructNode('3AxisTranslationManipulator', {
  targetNode: instance,
  size: 10,
  linearTranslationManipulators: true,
  planarTranslationManipulators: true,
  screenTranslationManipulators: true,
  undoManager: undefined // optionally you can specify it here, in case you have more than one
});
```

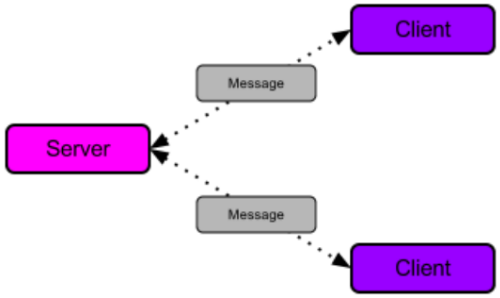
Chapter 16. Websockets Guide

The SceneGraph provides an easy way of using websockets to automate client - server - client communication. For this a node websocket server is required. For a sample implementation, please see our github project `http://github.com/fabric-engine/websocket-server`.

For test scenarios you might as well use node websocket server located at `ws.fabric-engine.com`.

16.1. WebSocketManager

Before you can send or receive messages you need to setup the **WebSocketManager**. Each session uses a unique key, which is determined automatically, however you can specify it yourself to ensure that all users of the application are able to communicate with each other.



Since the web socket connection has to be established before message handlers can be setup, the constructor also takes in the **onOpenCallBack** option, which is executed once the connection is up.

```
// create a web socket manager
var websocketMgr = scene.constructManager('WebSocketManager', {
  serverUrl: 'ws.fabric-engine.com',
  contextID: 'myOwnCustomKey',
  onOpenCallBack: function() {
    console.log('Connection established');
  }
});
```

16.2. Sending Messages

To send message, ensure that the connection has been established. Otherwise the sending of the message will fail. Messages contain a name, data as well as an optional recipient. If you don't specify the recipient the message will be send to all of the participants of the current session.

```
websocketMgr.sendMessage('askQuestion', { question: 'What\'s your name?' });
```

The data can contain any JSON compliant struct. The WebSocketManager encodes it and decodes it automatically.

16.3. Receiving Messages

To receive messages you need to setup callbacks for each message type on the WebSocketManager. Reception message callbacks themselves can also send messages again of course.

```
// setup a message callback for the 'askQuestion' message
```

```
websocketMgr.addMessageCallBack('askQuestion', function(message) {
  if(message.data.question == 'What\'s your name?') {
    // reply just to the sender of this message
    var data = {
      question: message.data.question,
      answer: 'My name is Fabric.'
    };
    websocketMgr.sendMessage('replyToQuestion', data, message.sourceID);
  } else
    console.log('I don\'t know how to answer this question: '+message.data.question);
});

// setup a message callback for the 'replyToQuestion' message
websocketMgr.addMessageCallBack('replyToQuestion', function(message) {
  console.log('I asked: '+message.data.question);
  console.log(message.sourceID+' replied: '+message.data.answer);
});
```

Note: The WebSockets.html sample application demonstrates how to synchronize camera manipulation between clients.

Chapter 17. Bullet Physics Guide

The Fabric Engine SceneGraph contains a reference implementation of the Bullet Physics simulation engine (`http://www.bulletphysics.org`). It allows to create several different shapes, rigid bodies, soft bodies as well as constraints between them.

Bullet requires custom shapes to be constructed for collision. They can be different from the ones you see in the viewport, so for example you could create a box shape for a complex mesh based on its bounding box for a faster simulation.

Bullet operates on the SceneGraph's transform nodes.

17.1. Creating the Bullet world

Each simulation requires a world to run in. You may create several worlds, but only object within the same world will effect each other.

```
var world = scene.constructNode('BulletWorldNode');
```

Once the world is set up you can start creating shapes and simulation elements.

17.2. Creating Collision Shapes

The SceneGraph's Bullet implementation supports several basic shapes. Shapes are identified by their name. The name has to be unique, using the same name several times will result in an error. You can later use the shapes' names to construct rigid bodies. These are the basic shapes you can construct:

```
world.addShape('myBox', FABRIC.RT.BulletShape.createBox(new FABRIC.RT.Vec3(1.0,1.0,1.0)));
world.addShape('mySphere', FABRIC.RT.BulletShape.createBox(1.0 /* radius */));
world.addShape('myCylinder', FABRIC.RT.BulletShape.createBox(1.0 /* radius */, 1.0 /* height
 */));
world.addPlane('myPlane', FABRIC.RT.BulletShape.createBox(new FABRIC.RT.Vec3(0.0,1.0,0.0)));
```

Aside from the standard shapes you can also construct complex shapes based on SceneGraph geometry nodes. So for example, given the torus triangles node, you can construct either a convex hull collision shape or a triangle mesh, or a GImpact shape. Convex hull shapes are very fast and robust, but they don't provide actual shape collision. triangle mesh shapes are very slow for moving objects, and should be used only for complex ground shapes. GImpact shapes provide accurate collision for moving objects which need to support actual shape collision (inclusing concave shapes).

```
var torus = scene.constructNode('Torus'); // this is a triangles node
world.addShape('myConvexHull',FABRIC.RT.BulletShape.createConvexHull(torus));
world.addShape('myTriangleMesh',FABRIC.RT.BulletShape.createTriangleMesh(torus));
world.addShape('myGImpact',FABRIC.RT.BulletShape.createGImpact(torus));
```

17.3. Creating Rigid Bodies

Once you have constructed the shapes, you can create rigid bodies in the world. Rigid bodies have several settings, but the main important option is the **mass**. If you set the mass to 0.0, the rigid body will be passive. This means Fabric Engine will continue to drive the transform of the rigid body, and it will affect the Bullet world. If the mass is set to > 0.0 the rigid body is active, and is purely affected by the Bullet simulation.

```
// construct a passive plane
world.addRigidBody('ground', new FABRIC.RT.BulletRigidBody({
  mass: 0.0
```

```
}), 'myPlane');

// construct a passive plane
world.addRigidBody('torusRbd1', new FABRIC.RT.BulletRigidBody({
  mass 1.0,
  friction: 0.6,
  restitution: 0.2, // bouncyness
  transform: new FABRIC.RT.Xfo({
    tr: new FABRIC.RT.Vec3(0.0,4.0,0.0)
  })
}), 'myConvexHull');
```

Note that we didn't connect up the rigid bodies to any element in the SceneGraph yet. Right now these objects are purely part of the world and are not visibile to the viewport at all. To create a transform that is connected to the Bullet world, you can construct a **BulletRigidBodyTransform** node. The code below will setup a torus which is moving and visible in the viewport.

Once you have constructed the shapes, you can create rigid bodies in the world. Rigid bodies have several settings, but the main important option is the **mass**. If you set the mass to 0.0, the rigid body will be passive. This means Fabric Engine will continue to drive the transform of the rigid body, and it will affect the Bullet world. If the mass is set to > 0.0 the rigid body is active, and is purely affected by the Bullet simulation. Also note that you can create multiple rigid bodies using the same collision shape.

```
// create a transform node driven by bullet
var bulletTransform = scene.constructNode('BulletRigidBodyTransform', {
  name: 'torusRbd2',
  bulletWorldNode: world,
  shapeName: 'myConvexHull',
  rigidBody: new FABRIC.RT.BulletRigidBody({
    transform: new FABRIC.RT.Xfo({
      tr: new FABRIC.RT.Vec3(0.0,4.0,0.0)
    })
  })
});

// create an instance to draw the torus
var instance = scene.constructNode('Instance', {
  transformNode: bulletTransform,
  geometryNode: torus,
  materialNode: scene.constructNode('PhongMaterial')
});
```

17.4. Creating Soft Bodies

While Rigid Bodies represent hard surfaces, Bullet also supports deforming surfaces, called **Soft Bodies**. These bodies contain the shape as well, so they don't require a collision shape name for construction. Soft bodies don't use a transform, they are applied as a deformation on top of the geometry node.

```
// create a softbody
var torusSoftBody = world.addSoftBody('torusSbd1',new FABRIC.RT.BulletSoftBody({
  trianglesNode: torus,
  transform: FABRIC.RT.xfo({
    tr: new FABRIC.RT.Vec3(0,16,0)
  }),
  stiffness: 0.1, // spring stiffness
  friction: 0.5, // dynamic friction
  conservation: 0.0, // volume conservation
  recover: 0.01, // recover original shape per frame
  pressure: 5    // internal pressure
}));

// create an instance to draw the deforming
// torus. no transform specified since the
```



```
// transform is included in the deformation
var instance = scene.constructNode('Instance', {
  geometryNode: torusSoftBody,
  materialNode: scene.constructNode('PhongMaterial')
});
```

17.5. Creating Constraints

Constraints are connections between rigid bodies. The SceneGraph supports three types of constraints:

- Point2Point: A ball and socket connection with 360 degrees of freedom
- Hinge: A single axis rotation connection with 360 degrees of freedom
- Slider: A single axis translation connection with 0 degrees of freedom

For creating a constraint you need to specify the names of the two bodies to be connected as well as the constraint's options.

```
world.addConstraint('myBallSocket',FABRIC.RT.BulletConstraint.createPoint2Point({
  pivotA: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,4.0,0)}),
  pivotB: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,-4.0,0)}),
}),'torusRbd1','torusRbd2');

world.addConstraint('myHinge',FABRIC.RT.BulletConstraint.createHinge({
  pivotA: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,4.0,0)}),
  pivotB: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,-4.0,0)}),
}),'torusRbd1','torusRbd2');

world.addConstraint('mySlider',FABRIC.RT.BulletConstraint.createSlider({
  pivotA: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,4.0,0)}),
  pivotB: FABRIC.RT.xfo({tr: new FABRIC.RT.Vec3(0,-4.0,0)}),
}),'torusRbd1','torusRbd2');
```

17.6. Creating Anchors

Anchors are connections between rigid bodies and soft bodies. Aside from the names of the rigid body and the soft body it is also required to specify an array of point indices on the softbody to attach the anchor to.

```
world.addAnchor('myAnchor',new FABRIC.RT.BulletAnchor({
  softBodyNodeIndices: [0,1,2,3,4,5]
  }),'torusRbd1','torusSbd1'
);
```

17.7. Array functionality

All **add** methods on the BulletWorldNode support single obbjects as well as arrays. This means that you can construct multiple rigid bodies at the same time, by providing an array of bodies rather than just a single one. This also works for constraints. For further details, please have a look at the *Constraints.html* or *StackingBoxes.html* sample applications.

Chapter 18. SceneGraphNode Reference

In this section you can find documentation for all of the current SceneGraphNodes provided by the JavaScript SceneGraph. For each node a short description is provided, as well as a reference to its options. Only the methods on the public interface are documented here. For further information on the private interfaces please refer to the JavaScript file providing the respective nodes.

18.1. SceneGraph.js

18.1.1. SceneGraphNode

ParentNode: **None**

This is the base node for all nodes in the SceneGraph. It is never constructed directly, but all other nodes use it as their base. The SceneGraphNode is responsible for constructing the dependency graph nodes and eventhandler nodes in the core, as well as deploying all of the base features provided by the SceneGraph, such as eventhandling etc.

Event related functions are only available on certain instances of the SceneGraphNode. This depends on the construction and if **addEventHandlingFunctions** was called.

Table 18.1. SceneGraphNode Options

name	The name of the SceneGraph node. If not specified, it will default to its type.

Table 18.2. SceneGraphNode Methods

getName()	Returns the name of the node.
setName(name)	Sets the new name of the node.
addEventListener(event, func)	Adds a new function to the given event name.
removeEventListener(event,func)	Removes a function from listening to a given event name.
fireEvent(event)	Executes all of the listening functions of a given event name.

18.1.2. ResourceLoad

ParentNode: **SceneGraphNode**

This node is used for loading external resources into Fabric Engine. Resources can be images, 3D files or any external binary content. The node supports loading the data to memory or storing it to a temporary file (default). The node also fires an *loadSuccess* event once the load is finished, so event listener can be attached accordingly.

Table 18.3. ResourceLoad Options

url	The url of the external data resource
storeDataAsFile	If true the data is stored to a temporary file, otherwise the content is stored in memory.
redrawOnLoad	If true the viewport will be refreshed automatically once loading is finished.

blockRedrawingTillResourceIsLoaded	If true the viewport will not refresh during the load of the resource.

Table 18.4. ResourceLoad Methods

setUrl(url, force)	Sets the new URL for the resource. If force is true the node is force to evaluate immediately.

18.2. Geometry.js

18.2.1. Geometry

ParentNode: **SceneGraphNode**

The Geometry node is a base abstract node for all geometry nodes. The Geometry node provides basic OpenGL Vertex Buffer Object (VBO) management and other services common to all type of geometry. The Geometry node provides a uniformsDependency Graph node which is used to contain non-vertex data for the geometry. Non vertex data might be paramters used in a generator opator such as the radius of a sphere. The Geometry node also provides a Vertex Attribute Dependency Graph node. All per-vertex data is stored in the vertex dgnode as a sliced node. Each slice in the sliced node contains the vertex data for one vertex. This special vertex attribute node ensures that every vertex has the same attributes, and by simply adding a new data member, all verticies aquire the new data mamber. This model maps well to GPU memory layouts where all vertex attributes must have identical counts to evaluate efficiently. This model enables efficient multi-threaded operators to be applied to all vertices in a mesh. The trade-off is lightly higher memory usage, as shared vertex attributes are duplicated to each vertex.

Each of the derrived geometries must provide specialiazed drawing and raycasting operators to enable the derived geometry to inetgrate with the rendering and interaction systems of the Fabric Scene Graph. The combination of flexible vertex attributes sets, and custom draw operator makes it easy to define custom geometry types.

Table 18.5. Geometry Options

createBoundingBoxNode	Determines whether a bounding box node is constructed.
drawable	Determines whether the redraw event hander is set up and VBOs uploaded for each vertex atribute. Note: drawing would not be required for a collision mesh for example
dynamicIndices	Specifies whether the indices buffer should be dynamic. Geometry which has a dynamcially chaning topoloy would set this to true.

Table 18.6. Geometry Methods

addUniformValue()	Adds a new member to the uniforms Dependency Graph node.
addVertexAttributeValue(name, type, attributeoptions)	Adds a new vertex attribute to the geometry. The options object passed in can specify ether the given vertex attribute should be used to generate a VBO using the parameter genVBO: true.
reloadVBO(name)	Manually triggers an re-uploading of the vertex buffer to the GPU.
setAttributeDynamic(name)	Sets the given attribute to use Dynamic OpenGL buffers. Not this should be called prior to the first time the geometry is drawn.
setAttributeStatic(name)	Sets the given attribute to use Static OpenGL buffers. Not this should be called prior to the first time the geometry is drawn.
getVertexCount	Sets the number of vertices stored in the attributes node.

setVertexCount	Gets the number of vertices stored in the attributes node.
loadGeometryData	This method is used to load data into the geometry node. The Collada parser uses this method to load parsed geometry data.s
getBoundingBox	Returns the corner coordinates of the bounding box for this node.s

18.2.2. GeometryDataCopy

ParentNode: **Geometry**

The geometry data copy node provides a methd to create geometries derived from a base geometry. Per-Point operators can be applied to the Geometry Data Copy, and these operators can reference the base geometry as an input. The GeometryDataCopy is a base type used to construct specialized deforming geometries. The GeometryDataCopy can be used to derive a new Point, Lines, or Triangle mesh as it only provides generic double buffering and operator evaluation of vertex attributes.

Table 18.7. GeometryDataCopy Options

baseGeometryNode	The geometry to use as the base for this derived geometry.

Table 18.8. GeometryDataCopy Methods

setBaseGeometry(geometry)	Sets the geometry to be used as base geometry.

18.2.3. Points

ParentNode: **Geometry**

The Points node creates a specialization of the gometry node for drawing point. It simply does this by adding a 'positions' vertex attribute, and providing drawing operators that issue the appropriate OpenGL draw comments during rendering.

18.2.4. Lines

ParentNode: **Geometry**

The Lines node creates a specialization of the gometry node for drawing lines. It simply does this by adding a 'positions' vertex attribute, and an indices Uniform value, and providing drawing operators that issue the appropriate OpenGL draw comments during rendering.

18.2.5. LineStrip

ParentNode: **Geometry**

The LineStrip node creates a specialization of the gometry node for drawing line strips. The only difference between Lines and LineStrip is the drawing operator provided assumes a GL_LINESTRIP layout of the vertex indices array.

18.2.6. Triangles

ParentNode: **Geometry**

The Triangles node creates a specialization of the gometry node for drawing triangles. It does this by adding several vertex attrinbutes that are used in drawing triangles. The 'positions' and 'normal' vertex attributes are provided, along with optional VU sets ('uvs0'), and tangents.

18.2.7. Instance

ParentNode: **SceneGraphNode**

The Instance Scene Graph node represents a drawn geometry. The Instance node is used to bind a Geomerty to a Transform and a Material and draw the results on screen. When a geometry is assigned to an instance node, the Instance node requests the geometries 'draw operator', and assigns the draw operator to the instance's draw event handler. The Instance node also handles raycasting of geometries. When a material is assigned to an Instance, the instance connectes its draw event handler to the materials draw event handler. This ensures that when the matierl is set up for rendering, the draw event handler of the Instance is then invoked causing the drawing of the geometry.

Table 18.9. Instance Options

transformNode	The transform node to use to control the transform the geometry in the scene.
transformNodeMember	TODO: remove me
transformNodeIndex	TODO: remove me
constructDefaultTransformNode	The Instance node will construct a default transform node if none is provided.
geometryNode	The Geometry node that will be used in drawing/racasting
transformNode	The Transform node that will be used in drawing/raycasting.
materialNode	The Material node that will be used in drawing.
enableRaycasting	The option to enable/disable raycasting for this instance
enableDrawing	The option to enable/disable drawing for this instance
raycastOverlaid	In some cases, it is required that a given geometry returns raycast results even if occlueded by other geometries. This is the case when a geometry is being used as a Manipulator widget. The Manipulator widgets are drawn over the top of all geometry, and must also respond to mouse events. This option causes the Instance to alsways return a raycast result with a distance of 0.

Table 18.10. GeometryDataCopy Methods

getGeometryNode/setGeometryNode	Gets/Sets the Geometry node
getTransformNode/setTransformNode	Gets/Sets the Transform node
getMaterialNode/setMaterialNode	Gets/Sets the Material node
getLayerManagerNode/setLayerManagerNode	Gets/Sets the Layer Manager node. When an Instance is assigned to a layer in a Layer Manager, it binds an operator to its dgnode that drives the draw toggle from the value of the assigned layer in the layer manger. This means that if the layer is disabled, all Instances assigned to that layer become disabled.

18.2.8. LayerManager

ParentNode:

The Layer Manager is a very simply Manager for contolling the drawing of many Instances at once. When an I

Table 18.11. LayerManager Methods

addLayer(geometry)	Adds a new Boolean value to the Layer Manger with the given name. Each assigned layer generates getter and setter functions to toggle the state of the drawing for that layer.

18.3. Primitives.js

The Primitives are generated geometries, using operators to procedurally generate the data of the geometry. They expose methods to access and modify the paramters used by the generator operators.

18.3.1. LineVector

ParentNode: **Lines**

The LineVector node simply draws a static line between 2 predefined points.

Table 18.12. LineVector Options

from	The start point of the line in the coordinte space of the line.
to	The end point of the line in the coordinte space of the line.

18.3.2. Cross

ParentNode: **Lines**

The Cross node uses an operator to control the size and position of the 6 points that make up the 3 orthogonal line segments that make up the cross. The cross is drawn as 3 line degments aligned to the majior axes of the local coordinate frame.

Table 18.13. Cross Options

size	The size of the generated cross.

Table 18.14. Cross Methods

getSize/setSize	Accessor methods for the size of the cross

18.3.3. Axis

ParentNode: **Lines**

The Axis node uses an operator to draw a set of labeled coordinate system axis. The axes are labeled 'X', 'Y', and 'Z'

Table 18.15. Axis Options

size	The size of the generated cross.

Table 18.16. Axis Methods

getSize/setSize	Accessor methods for the size of the cross

18.3.4. Rectangle

ParentNode: **Lines**

The Rectangle primitive displays a rectangle aligned to the X/Z plane.

Table 18.17. Rectangle Options

length	The length of the generated rectangle in the local X axis.
width	The width of the generated rectangle in the local Z axis.

Table 18.18. Rectangle Methods

getLength/setLength	Accessor methods for the length of the rectangle primitive
getWidth/setWidth	Accessor methods for the width of the rectangle primitive

18.3.5. BoundingBox

ParentNode: **Lines**

The BoundingBox primitive displays a box using top left, and bottom right coordinates to control the size and offset of the box.

Table 18.19. BoundingBox Options

bboxmin	The length of the generated rectangle in the local X axis.
bboxmax	The width of the generated rectangle in the local Z axis.

Table 18.20. BoundingBox Methods

getBoundingBoxMin/setBoundingBoxMin	Accessor methods for the min coordinates of the bounding box.
getBoundingBoxMax/setBoundingBoxMax	Accessor methods for the max coordinates of the bounding box.

18.3.6. Grid

ParentNode: **Lines**

The Grid primitive displays a 3 dimensional grid

Table 18.21. Grid Options

size_x	The size of the grid in the X axis
size_y	The size of the grid in the Y axis
size_z	The size of the grid in the Z axis
sections_x	The number of sections of the grid along the X axis
sections_y	The number of sections of the grid along the Y axis
sections_z	The number of sections of the grid along the Z axis

18.3.7. CameraPrimitive

ParentNode: **Lines**

The CameraPrimitive is used to draw a Camera icon on screen. The camera primitive draws a box with a cone attached indicating the oriantation of the camera. This can be used to visualize cameras in the 3d viewport.

Table 18.22. CameraPrimitive Options

size	The size of the generated cross.

Table 18.23. CameraPrimitive Methods

getSize/setSize	Accessor methods for the size of the drawn primitive

18.3.8. Circle

ParentNode: **Lines**

The Circle primitive can be used to draw either a circle, or an arc if a the arc ancle specified is less than 2pi.

Table 18.24. CameraPrimitive Options

radius	The size of the generated cross.
arcAngle	the angle, specified in radians of the drawn Circle. By default the 'Circle' draws a complete circle, however, an arc can be drawn by specifying a value less than 2 pi.
numSegments	The number of line segments used to draw the circle/arc.

Table 18.25. CameraPrimitive Methods

getRadius/setRadius	Accessor methods for the radius of the drawn circle/arc
getArcAngle/setArcAngle	Accessor methods for the angle of the drawn circle/arc
getNumSegments/setNumSegments	Accessor methods for the number of line segments use when drawing the arc/circle

18.3.9. Plane

ParentNode: **Triangles**

The Plane primitive draws

Table 18.26. Plane Options

length	The length of the plane in the X axis
width	The width of the plane in the Z axis
lengthSections	The number of length sections to use when building the plane.
widthSections	The number of width sections to use when building the plane

Table 18.27. Plane Methods

getLength/setLength	Accessor methods for the length of the drawn plane
getWidth/setWidth	Accessor methods for the width of the drawn plane

getLengthSections/setLengthSections	Accessor methods for the number of length sections to use when drawing the plane.
getWidthSections/setWidthSections	Accessor methods for the number of width sections to use when drawing the plane.

18.3.10. Cuboid

ParentNode: **Triangles**

The Cuboid primitive draws a cuboid.

Table 18.28. Cuboid Options

length	The length of the cuboid in the X axis
width	The width of the cuboid in the Z axis
height	The height of the cuboid in the Y axis
lengthSections	The number of length sections to use when building the cuboid.
widthSections	The number of width sections to use when building the cuboid
heightSections	The number of height sections to use when building the cuboid

Table 18.29. Cuboid Methods

getLength/setLength	Accessor methods for the length of the drawn cuboid
getWidth/setWidth	Accessor methods for the width of the drawn cuboid
getLengthSections/setLengthSections	Accessor methods for the number of length sections to use when drawing the cuboid.
getWidthSections/setWidthSections	Accessor methods for the number of width sections to use when drawing the cuboid.
getHeightSections/setHeightSections	Accessor methods for the number of height sections to use when drawing the cuboid

18.3.11. Sphere

ParentNode: **Triangles**

The Sphere primitive draws a sphere on screen using triangles.

Table 18.30. Sphere Options

radius	The radius of the sphere
detail	The detail parameter controls the number of length and width sections.

Table 18.31. Sphere Methods

getRadius/setRadius	Accessor methods for the length of the sphere
getDetail/setDetail	Accessor methods for the width of the sphere

18.3.12. Torus

ParentNode: **Triangles**

The Torus primitive draws a torus on screen using triangles.

Table 18.32. Torus Options

innerRadius	The radius of the hole throught he middle of the torus
outerRadius	The radius of the body section of the torus
detail	The detail parameter controls the number of sections used to generate the shape of the torus.

Table 18.33. Torus Methods

getInnerRadius/setInnerRadius	Accessor methods for the length of the torus
getOuterRadius/setOuterRadius	Accessor methods for the outer radius of the torus
getDetail/setDetail	Accessor methods for the detail paramerer of the torus

18.3.13. Cone

ParentNode: **Triangles**

The Torus primitive draws a torus on screen using triangles.

Table 18.34. Torus Options

innerRadius	The radius of the hole throught he middle of the Cone
outerRadius	The radius of the body section of the Cone
detail	The detail parameter controls the number of sections used to generate the shape of the Cone.

Table 18.35. Torus Methods

getInnerRadius/setInnerRadius	Accessor methods for the length of the Cone
getOuterRadius/setOuterRadius	Accessor methods for the outer radius of the Cone
getDetail/setDetail	Accessor methods for the detail paramerer of the Cone

18.3.14. Cylinder

ParentNode: **Triangles**

The Cylinder primitive draws a cylinder on screen using triangles.

Table 18.36. Cylinder Options

radius	The radius of the body of the Cylinder
height	The height of the body of the Cylinder
caps	Constrols whether the generated cylinder is capped at both ends.
sides	The number of sides to use when generating the cylinder.

loops	the number of loops to use when generating the cylinder.

Table 18.37. Cylinder Methods

getRadius/setRadius	Accessor methods for the radius of the Cylinder
getHeight/setHeight	Accessor methods for the height of the Cylinder
getCaps/setCaps	Accessor methods for the caps parameter of the Cylinder
getSides/setSides	Accessor methods for the number of sides of the Cylinder
getLoops/setLoops	Accessor methods for the number of loops of the Cylinder

18.3.15. Teapot

ParentNode: **Triangles**

The Teapot primitive draws the classic teapot primitive onscreen using triangles.

Table 18.38. Teapot Options

size	The size of the teapot primitive
detail	The detail parameter controls the number of sections used to generate the shape of the Teapot.

Table 18.39. Teapot Methods

getSize/setSize	Accessor methods for the length of the Teapot
getDetail/setDetail	Accessor methods for the detail paramerer of the Teapot

18.4. Cameras.js

18.4.1. Camera

ParentNode: **SceneGraphNode**

The base Camera node is used to control the viewpoint and projection used during rendering.

Table 18.40. Camera Options

nearDistance	The near clipping plane distance for the camera.
farDistance	The far clipping plane distance for the camera.
fovY	The vertical (Y) field of view angle for this camera.
focalDistance	The focal distance for the camera.
orthographic	Set to true the camera is rendered in orthographic more, otherwise perspective mode is used.
transformNode	The transform node to use.
screenOffset	Viewport center offset

Table 18.41. Camera Methods

getNearDistance/setNearDistance	Get/Set the near clipping plane distance.

getFarDistance/setFarDistance	Get/Set the far clipping plane distance.
getFovY/setFovY	Get/Set the vertical (Y) field of view angle.
getFocalDistance/setFocalDistance	Get/Set the focal distance. Note: the focal distance may be used in shaders that perform lense effects. Also, the focal distance is computed in the 'TargetCamera' defined below.
getOrthographic/setOrthographic	Sets the given attribute to use Static OpenGL buffers. Not this should be called prior to the first time the geometry is drawn.
getTransformNode/setTransformNode	Get/Set the Transform node used by this camera.

18.4.2. FreeCamera

ParentNode: **Camera**

A free camera is transformed using a regular Transform node, rather than using a target position to align the camera. A free camera is usefull when setting up orthographic projections where you do not want the camera to alight ot any given target, but simpy maintain a given orientation. The Free camera exposes no extra methos, and simply configures a default transform node.

Table 18.42. FreeCamera Options

position	The initial position of the camera.
orientation	the initial orientation of the camera.

18.4.3. TargetCamera

ParentNode: **Camera**

The TargetCamera uses the 'AimTransform' node to align itself with a given target. The TargetCamera uses the distance from the camera to the target to compute its focalDistance. Most of the demos in the Fabric demos page use Target Camera as it it

18.5. Images.js

The nodes defined in the Images file are all retating to the storage of pixel data, either from loaded images, or generated on the GPU.

18.5.1. Image

ParentNode: **SceneGraphNode**

The base Image provides no functionality and is simply a parent type for all image types.

18.5.2. RenderTargetBufferTexture

ParentNode: **Image**

The RenderTargetBufferTexture is used to bind render target buffers to be used as textures. This Image type is used for planar reflections, where the scene is rendered to a buffer and then the buffer used as a texture on a reflective surface. It is also used in the Deffered Renderer to provide the buffered generated in the 'pre-pass' as textures to the 'post pass' shaders.

Table 18.43. RenderTargetBufferTexture Options

bufferIndex	The index of the render target buffer to be bound as a texture.

18.5.3. Image2D

ParentNode: **Image**

The Image2D node is the basic texture storage node. The Image2D node holds generic image data (members: pixels, width, height), which might be color or grayscale. If 'options.createResourceLoadNode' is passed in as true, an URL-based image loader will be incorporated, and used to load the pixel data. The Image2D node currently supports RGB, RGBA 8 bits per pixel images, and Color 32 bits per pixel images, and Scalar pixel formats.

Table 18.44. Image2D Options

format	Pixel format. Currently supported: RGB, RGBA, Color and Scalar.
createDgNodes	If this is set to true the Image node will construct a dgnode to store the pixel data. The dgnode stores the data in multiple slices, one slice for each pixel. This is usefull for when building applications that need to perform per-pixel operations on the CPU. operators applied to the pixels dgnode operate in a similar way to operators that operate on vertices of a geometry, being applied to many pixels in parallel.
createResourceLoadNode	Set to true this flag will enable the Image node to load a texture off a resource load node.
createLoadTextureEventHandler	If the image uses a ResouceLoadNode and this flag is set, it will create an EventHandler for the Image being loaded.
width	The width of the empty Image
height	The height of the empty Image
color	When initializing an emply image, all the pixels are set to this color
url	
forceRefresh	If this is set, the Image will always be re-loaded onto the GPU. This is useful for animated Images.
glRepeat	If set to true, the image is set to repeat in bot U and V directions.

Table 18.45. Image2D Methods

getWidth	Get the width of the stored texture. Note: this method is only available when createDgNodes is set to true when constructing this node.
getHeight	Get the height of the stored texture. Note: this method is only available when createDgNodes is set to true when constructing this node.
getResourceLoadNode	Get the resource load node used to load the image data.
isImageLoaded	Returns true once the image has finished loading.

18.5.4. Image3D

ParentNode: **Image**

The Image3D node holds generic image data (members: pixels, width, height, depth), which might be color or grayscale. If 'options.createResourceLoadNode' is passed in as true, an URL-based image loader will be incorporated, and used to load the pixel data. The Image3D node currently supports RGBA 8 bits per pixel images, and Color 32 bits per pixel images. The Image3D node is used in the volume rendering demos to store the volumetric MRI data.

Table 18.46. Image2D Options

format	Pixel format. Currently supported: RGB, RGBA, Color and Scalar.
createDgNodes	If this is set to true the Image node will construct a dgnode to store the pixel data. The dgnode stores the data in multiple slices, one slice for each pixel. This is usefull for when building applications that need to perform per-pixel operations on the CPU. operators applied to the pixels dgnode operate in a similar way to operators that operate on vertices of a geometry, being applied to many pixels in parallel.
createResourceLoadNode	Set to true this flag will enable the Image node to load a texture off a resource load node.
createLoadTextureEventHandler	If the image uses a ResouceLoadNode and this flag is set, it will create an EventHandler for the Image being loaded.
width	The width of the empty Image
height	The height of the empty Image
depth	The depth of the empty Image
url	
glRepeat	If set to true, the image is set to repeat in bot U and V directions.

Table 18.47. Image2D Methods

getWidth	Get the width of the stored texture. Note: this method is only available when createDgNodes is set to true when constructing this node.
getHeight	Get the height of the stored texture. Note: this method is only available when createDgNodes is set to true when constructing this node.
getDepth	Get the depth of the stored texture. Note: this method is only available when createDgNodes is set to true when constructing this node.
getResourceLoadNode	Get the resource load node used to load the image data.
isImageLoaded	Returns true once the image has finished loading.
getUrl/setUrl	interface to the URL used to load the pixel data for this image.

18.5.5. CubeMap

ParentNode: **Image**

The CubeMap node contains 6 Image nodes which can be used to texture with cubic mapping. The 6 sides of the cube are loaed as separate textures. Note: the CubeMap only supports 8 bits per pixel textures at this point.

Table 18.48. CubeMap Options

urls	An array of six URLs to the six images to load.

18.5.6. Video

ParentNode: **Image**

The CubeMap node contains 6 Image nodes which can be used to texture with cubic mapping. The 6 sides of the cube are loaed as separate textures. Note: the CubeMap only supports 8 bits per pixel textures at this point.

Table 18.49. Video Options

url	The url of the video to load.
animationControllerNode	The animation controller used to drive the playback of the video

Table 18.50. Image2D Methods

getAnimationController/setAnimationController	Get or set the Animation Controller. Animation Controllers are used to calculate the time values to be used in the video node. The animation controller can be used to calculate a looping time sequence for example.

18.5.7. ScreenGrab

ParentNode: **SceneGraphNode**

The ScreenGrab node is used to capture the rendered openGL viewport to an image file.

Table 18.51. Image2D Methods

saveAs	Used to open a saveAs dialog box and write out the image as a PNG file.

18.6. Materials.js

The materials system in Fabric is built using OpenGL shader based rendering using GLSL shaders.

18.6.1. Shaders vs Materials

The rendering system in Fabric is built around the depth first traversial of the event hander graph atached to the window redraw event. The redraw event graph represents a callstack, where at each event handler, the children are visited in order and executed. A parent nodes is always evaluated before a child node, and a parent node can have many children. The 'Shader' node is the parent of the 'Material' nodes attached to it. The Shader node is responsible for loading the GLSL shader code into the GPU, and setting the shader context. The Material node is responsible for setting shader parameters, such as setting color values, or binding textures.

Changing the shader constex is a relatively expensive operation and should be minimized, and this is what drove the design of the rendering pipeline. For each shader, we load the shader and set the context, then for each Material, we set the parameters, and load appropriate textures. This graph structure enables batching of rendered objects acording to the Shader that they use, and also the Material parameters. This means that if you have many objects all drawn using the same shader and material parameters,they will be drawn together using only one shader binding.

18.6.2. getShaderParamID

The getShaderParamID function is used to generate unique identifier IDs for shader attribute and uniform names. For example, a shader might have a uniform named 'positions'. At graph construciton time, we convert the string 'positions' into a number so that we can match shader uniforms with data in the graph without doing many string comparisons each frame. At the time the rendering system was being developed, KL did not provide strong support for strings, however this has changed, and so this techniqure may be removed in future.

18.6.3. Shader

ParentNode: **SceneGraphNode**

The Shader node is used to load a GLSL shader onto the GPU. Shader nodes are never constructed directly, and instead are constructed by the Material. Each type of Material shares a common shader node. The Scene maintains a map of constructed Shader nodes, ensuring that only one of each type is constructed. This means that for a given Material type, such as 'FlatTextureMaterial', even if many different instances of the materil are constructed using different textures, only one shader node is constructed, and loaded during rendering.

18.6.4. Material

ParentNode: **SceneGraphNode**

The Material node is the base node used to construct all types of Materials. The Material node uses many passed in options to define the material interface. The material system is driven using XML files.

18.6.4.1. Material XML File Structure

The Material node is the base node used to construct all types of Materials. The Material node uses many passed in options to define the material interface. The material system is driven using XML files.

The XML files define a mapping between GLSL shader code, and the naming conventions used in the SceneGraph. e.g. in your GLSL shader code you could use a variable called 'u_mvp', which contains your model view projeciton matix. In Fabric, the name this value is 'modelViewProjectionMatrix'. The XML file enables you to specify the mapping between your own variable names, and the naming convention used in Fabric. For variables that do not have a predefined meaning the the scenegraph, such as custom parameters for shaders, the XML file neables providing a user friendly name to be used to generate getters and setters for these paramers in JavaScript.

XML files containing shaders can be used to define custom material nodes using the helper function:

```
FABRIC.SceneGraph.defineEffectFromFile('MaterialName', 'MaterialXMLFile.XML);
```

The genrated node exposes and interface for assigning uniform values such as color and opacoity values, and also exposes an interface for assigning references such as textures and lights.

18.6.4.1.1. name

The name parameter is currently not used.

18.6.4.1.2. prototypeMaterialType

The prototypeMaterialType parameter enables the specification of a base material node to derive this custom material from. There are a collection of base material types that provide custom functionality that custom materials can inherit from.

18.6.4.1.3. uniforms

The uniforms section defines the mapping between the uniforms used in the GLSL shader code, and the the interface exposed by the material.

18.6.4.1.3.1. name

The name value specifies the name used int he GLSL code. This name value can be different for each shader, and will often be shortened or abbreviated names used in the glsl code.

18.6.4.1.3.2. constant

The constant value specifies the lable used in the SceneGraph. For example, if the constant name for the uniform 'u_materialColor' is specified as 'color', then a member called 'color' will be added to the material node, and a getter function called 'getColor', and a setter function called 'setColor' will be defined.

18.6.4.1.3.3. type

The type value specifies the Fabric registered type to use for this constant. All the GLSL types have corresponding Fabric types that are used when loading the uniform values.

18.6.4.1.3.4. owner

The owner value is optional. The owner attribute specifies that the Material should not generate a member and getter and setter funcions, because this uniform is loaded by the specified node. For example, when the owner is listed as 'instance', the instance will load that uniform, and the material should not provide loading functions.

18.6.4.1.4. attributes

The attributes section defines the mapping between the attributes used in the GLSL shader code, and the vertex attributes stored in the geometry node. every shader uses the 'positions' attribute, but any custom vertex attributes can be listed, and if they exist on the geometry, they will be loaded prior to shader invocation.

18.6.4.1.4.1. name

The name value specifies the name used in the GLSL code. This name value can be different for each shader, and will often be shortened or abbreviated names used in the glsl code.

18.6.4.1.4.2. binding

The binding value specifies the name of the member on the geometry's attribute node. For example, all triangles support 'positions', and 'normals' as attributes, but geometries can have any number of attributes assigned, and those attributes can be mapped to GLSL shader attributes using the binding value here.

18.6.4.1.5. programParams

The programParams sections enables the definition of custom GLSL program parameters. Any OpenGL program params can be listed, and can be used, for example, to control tesselation, or geometry shader parameters. An example of a shader that uses custom program params, is the 'NormalShader' that uses the OpenGL 'Geometry Shader' to convert vertex positions and normals to lines to enable drawing of normals on screen.

18.6.4.1.5.1. name

The name value specifies the name of the OpenGL program param.

18.6.4.1.5.2. value

The value attribute specifies the value to set the specified OpenGL program param to.

18.6.4.1.6. drawParams

The drawParams sections enables the definition of custom GLSL draw parameters that are to be set by the shader node.

18.6.4.1.6.1. drawMode

The drawMode can be set to a custom draw method, instead of using the default fo the drawn geometry type.

18.6.4.1.6.2. patchVertices

The patchVertices value can be specified when using tesselation shaders to control how the patches are generated at the tesselation stage.

18.6.4.1.7. Shader Sources

For a complete OpenGL shader to run, shader sources must be specified for each relevant shader stage. In the Shader sources of the XML file, source code can be specified for vertex, and fragment shaders, and also the more advanced shaders such as Geometry, and Tesselation shaders.

18.6.4.1.7.1. include

Each of the various shader sources can include extra files. These included files are simply inlined and evaluated as part of the entire shader source.

18.6.4.1.7.2. vertexshader

The vertex shader sectionb lists GLSL vertex shader code.

18.6.4.1.7.3. fragmentshader

The fragment shader sectionb lists GLSL fragment, or pixel shader code.

18.6.5. ShadowMapMaterial

ParentNode: **Material**

The ShadowMapMaterial is a base material type that enables materials that are used in shadow map rendering to be derived from. The ShadowMapMaterial integrates the material into the drawing pipline at the point where shadow maps are being rendered from the lights point of view. The

18.6.6. PointMaterial

ParentNode: **Material**

The PointMaterial extends the base Material with parameters for controlling the size of drawn points using the Fixed-Function OpenGL calls. By specifying the PointMaterial as the 'PrototypeMaterial' in the shader XML file, shaders can extend this material type with custom drawing code.

Table 18.52. PointMaterial Options

pointSize	The default size value to be used to draw the points

Table 18.53. PointMaterial Methods

getPointSize/setPointSize	Get and set the size values used to render the points.

18.6.7. LineMaterial

ParentNode: **Material**

The LineMaterial extends the base Material with parameters for controlling the thickness of drawn lines using the FixedFunction OpenGL calls. By specifying the LineMaterial as the 'PrototypeMaterial' in the shader XML file, shaders can extend this material type with custom drawing code.

Table 18.54. LineMaterial Options

lineWidth	The default thickness value to be used when drawing the lines

Table 18.55. LineMaterial Methods

getPointSize/setPointSize	Get and set the size values used to render the points.

18.6.8. TransparentMaterial

ParentNode: **Material**

The TransparentMaterial is a base material type that enables transparent materials to be integrated in the drawing pipeline. The TransparentMaterial integrates the material into the drawing pipline after all opaque materials have been drawn.

18.6.9. InstancingMaterial

ParentNode: **Material**

The InstancingMaterial is a base material type that is used when drawing multiple instances of the same geometry. The InstancingMaterial sets the paramter on the shader program called numInstances. The InstancingMaterial is used in conjunction with TransformTexture. The Texture is used to load an array of model matricies, and the Materials tells the shader how many insances to draw.

18.6.10. PostProcessEffect

ParentNode: **Material**

The PostProcessEffect is a base material type for post processing effect Material nodes. The PostProcessEffect is attached to the Viewport, and is bound before any of the scenes geometry is rendered. The PostProcessEffect material sets up an Frame Buffer Object and binds it during rendering before rendering traverses to the geometries. On the post descend operator stack, it unbinds the FBO, and binds it as a texture for the shader. A full screen quad is then rendered using the shader. There are several example post processing effect shaders provided that show how the post processing effect integrates with the render pipeline.

18.7. Animation.js

18.7.1. AnimationContainer

ParentNode: **SceneGraphNode**

The AnimationContainer is a base type for several kinds of animation containers. The Animationcontainer provides functionality common to all types of animation containers. Each animation container can contain only one type of keyframe. For example, most animation containers store Scalar keyframes, and therefore return a Scalar value during evaluation. However, AnimationContainers can also store 'Color', or 'Quat' values.

Table 18.56. AnimationContainer Options

controllerNode	The controller node used to control the evaluated time value can be passed into the constructor.

Table 18.57. AnimationContainer Methods

getValueType	Returns the data type that this animation container evaluates to.

18.7.2. TrackAnimationContainer

ParentNode: **AnimationContainer**

The TrackAnimationContainer stores a set of tracks of the same type. The TrackAnimationContainer is a base type for all 'Track Containers'.

Table 18.58. TrackAnimationContainer Options

keyframetype	The type of keyframe this animation container will store.

Table 18.59. TrackAnimationContainer Methods

getTimeRange	Returns the time range for all keys stored in all tracks.
getTrack/setTrack	Gets or sets an individual track.
getTracks/setTracks	Gets or sets all the tracks in the container.
getTrackCount	Returns the number of tracks the container holds.
setValues	Used by the manipulation system. Used to create keyframes on the tracks by setting new values at a given time.
bindNodeMembersToTracks	The tracks stored in a track container can be bound to memebrs on nodes in the graph. This function generates an operator that evaluates the given tracks and setting the bound member to the returned value.
openCurveEditor	Opens the curve editor to display the tracks in the container.

18.7.3. CharacterAnimationContainer

ParentNode: **AnimationContainer**

The CharacterAnimationContainer stores multiple sets of tracks of the same type. The CharacterAnimationContainer is a base type for track containers used in character animation. Each slice of the Dependency Graph Node stores a complete set of animation tracks. This enables a single CharacterAnimationContainer node to store a library of character animations. Storing all the character animations in a single node enables a character to dynamicaly select which set of animations it is evaluating, enabling such things as non-linear animation editing, animation blending, and crowd simulation.

Table 18.60. CharacterAnimationContainer Options

keyframetype	The type of keyframe this animation container will store.

Table 18.61. CharacterAnimationContainer Methods

addTrackSet	Adds a new track set to the container. The bindings used to evaluate the track set must also be provided.
getTimeRange	Returns the time range for all keys stored in all tracks.
getTrack/setTrack	Gets or sets an individual track.
getTracks/setTracks	Gets or sets all the tracks in the container.
getTrackSetCount	Returns the number of track sets stored in the container.
setValues	Used by the manipulation system. Used to create keyframes on the tracks by setting new values at a given time.
bindToRig	Binds the animation tracks to a rig. This function is used when generating keyframes for a rig, based on the motion of a different rig.
plotKeyframes	Once an animation container is bound to a rig, the tracks can be plotted using leanrly spaced keyframes.
openCurveEditor	Opens the curve editor to display the tracks in the container.

Table 18.62. CharacterAnimationContainer Methods

addTrackSet	Adds a new track set to the container. The bindings used to evaluate the track set must also be provided.
getTimeRange	Returns the time range for all keys stored in all tracks.
getTrack/setTrack	Gets or sets an individual track.
getTracks/setTracks	Gets or sets all the tracks in the container.
getTrackSetCount	Returns the number of track sets stored in the container.
setValues	Used by the manipulation system. Used to create keyframes on the tracks by setting new values at a given time.
bindToRig	Binds the animation tracks to a rig. This function is used when generating keyframes for a rig, based on the motion of a different rig.
plotKeyframes	Once an animation container is bound to a rig, the tracks can be plotted using leanrly spaced keyframes.
openCurveEditor	Opens the curve editor to display the tracks in the container.

18.7.4. LinearTrackAnimationContainer

ParentNode: **TrackAnimationContainer**

The LinearTrackAnimationContainer stores tracks or type LinearKeyframe.

18.7.5. LinearCharacterAnimationContainer

ParentNode: **CharacterAnimationContainer**

The LinearCharacterAnimationContainer stores tracks or type LinearKeyframe.

18.7.6. BezierTrackAnimationContainer

ParentNode: **TrackAnimationContainer**

The LinearCharacterAnimationContainer stores tracks or type BezierKeyframe.

18.8. Persistence.js

18.8.1. SceneSerializer

ParentNode: **SceneGraphNode**

The SceneSerializer manages the persistence of objects. Scene Graph nodes are added to the SceneSerializer, and the serializer invokes the weriteData methods on each node. The SceneSerializer also manages storing nodes in order of thier dendencies. By adding a node, all of that nodes dependencies are also added to the SceneSerializer, unless they are filtered by the type, or name filters.

Table 18.63. SceneSerializer Options

filteredNodeTypes	An Array of strings specifying node types that should not be saved by the serializer. This is usefull for avoiding saving such things as Cameras and Manipulators to your persisted storage.

Table 18.64. SceneSerializer Methods

addNode	Add a node to the SceneSerializer to be stored.
filterNode	Filter this node, ensuring it will not be saved.
serialize	Invoke the serialization of all added nodes.
save	Write the serialized data to the given storage system.

18.8.2. SceneDeserializer

ParentNode: **SceneGraphNode**

The SceneDeserializer loads data from a given storage medium, and constructs node, and loads thier data.

Table 18.65. SceneDeserializer Options

preLoadScene	If set to true, the all the nodes in the current scene are preloaded.
filteredNodeTypes	An Array of strings specifying node types that should not be loaded by the deserializer. This is usefull for avoiding loading such things as Cameras and Manipulators to your persisted storage.

Table 18.66. SceneDeserializer Methods

setPreLoadedNode	node	Preload a node into the SceneDeserializer. During deserialization, if a node has been preloaded, and data for a stored node matches the preloaeded node, then the data is loaded onto the preloaded node. This is usefull for loading presets into existing scenes.
load	storage, callback	Passing a storage interface, and a callback, the load function retrieves the data from the storage, and constructs the graph and/or loads dta into the preloaded nodes.

18.8.3. LogWriter

The LogWriter is a storage inteface used only in debugging. Instead of storing the data generated by the SceneSerializer, it simply logs the data to the console.

18.8.4. LocalStorage

The LocalStorage is used to write data to the HTML5 local storage supported in modern browsers. This usefull when the size of the data is small, and also does not require a confimation from the user.

18.8.5. FileWriter

The FileWriter is a storage inteface used by the scene serializer to write out JSON text files to the users hard drive.

18.8.6. FileWriterWithBinary

The FileWriterWithBinary is a storage inteface used by the scene serializer to write out JSON text files to the users hard drive, along with an accompanying binary data file. The binary data file ise valuable when the size of the data is very large. The binary data file is written using ZLib to compress the data, resulting in compact data files.

18.8.7. FileReader

The FileWriter is a storage inteface used by the scene deserializer to load and read JSON text files from the users hard drive.The Fire reader prompts the user to pick a file on construction.

18.8.8. XHRReader

The XHRReader is a storage inteface used to retrieve data via URLs through XMLHttpRequests.

