Broadcom Scene Graph

Contents

[Introduction 5](#_Toc349550570)

[Namespaces and Naming 5](#_Toc349550571)

[Terminology 5](#_Toc349550572)

[3D Backgrounder 6](#_Toc349550573)

[Coordinate Systems 6](#_Toc349550574)

[Transformations 7](#_Toc349550575)

[Color 7](#_Toc349550576)

[Primitives 7](#_Toc349550577)

[Rasterisation 7](#_Toc349550578)

[OpenGL ES 2.0 Pipeline 9](#_Toc349550579)

[Anatomy of an Application 10](#_Toc349550580)

[Application Options 10](#_Toc349550581)

[Platform 11](#_Toc349550582)

[Application 11](#_Toc349550583)

[Introducing the Scene Graph 13](#_Toc349550584)

[Inside Scene Nodes 14](#_Toc349550585)

[Example Scene Graph 16](#_Toc349550586)

[Notes on Handles 17](#_Toc349550587)

[Hello BSG 18](#_Toc349550588)

[Building BSG under BSEAV 18](#_Toc349550589)

[Building BSG for Trellis 19](#_Toc349550590)

[API Documentation 19](#_Toc349550591)

[Hello Cube 1 – Basic Setup 20](#_Toc349550592)

[Hello Cube 2 – Moving Stuff About 22](#_Toc349550593)

[Hello Cube 3 – Adding Texture 24](#_Toc349550594)

[Hello Cube 4 – Using the Hierarchy 25](#_Toc349550595)

[Planets 27](#_Toc349550596)

[Textures and Images 27](#_Toc349550597)

[Geometry 27](#_Toc349550598)

[Animation Details 30](#_Toc349550599)

[Extrapolation 32](#_Toc349550600)

[Going Deeper into the Scene Graph 34](#_Toc349550601)

[Constraints 34](#_Toc349550602)

[Callbacks 36](#_Toc349550603)

[Materials 36](#_Toc349550604)

[Scene Nodes 37](#_Toc349550605)

[Cameras 38](#_Toc349550606)

[Level of Detail 38](#_Toc349550607)

[Multi-threading in BSG 39](#_Toc349550608)

[Delay Loading Textures 39](#_Toc349550609)

[Asynchronous Callbacks 40](#_Toc349550610)

[Mutex 40](#_Toc349550611)

[Event 40](#_Toc349550612)

[Synchronous Callbacks 40](#_Toc349550613)

[Overview of Effect Files 41](#_Toc349550614)

[Options 41](#_Toc349550615)

[Passes 42](#_Toc349550616)

[Fonts and Text 43](#_Toc349550617)

[Direct 2D Text 44](#_Toc349550618)

[Scene Graph Text 45](#_Toc349550619)

[Modifying Text Behavior 46](#_Toc349550620)

[Off-screen Rendering 46](#_Toc349550621)

[Framebuffer Discard 48](#_Toc349550622)

[Mathematical and Utility Classes 48](#_Toc349550623)

[Vectors 48](#_Toc349550624)

[Matrices 48](#_Toc349550625)

[Quaternions 49](#_Toc349550626)

[Transforms 49](#_Toc349550627)

[Exceptions 50](#_Toc349550628)

[Time 50](#_Toc349550629)

[Video 51](#_Toc349550630)

[GUI Classes 53](#_Toc349550631)

[Carousel 53](#_Toc349550632)

[BSG Hints and Tips 54](#_Toc349550633)

[Assemblies 54](#_Toc349550634)

[Performance Tips 55](#_Toc349550635)

[Extending BSG 55](#_Toc349550636)

[Custom Geometry (basic) 56](#_Toc349550637)

[Adding Animation Types (intermediate) 56](#_Toc349550638)

[Derived Handles (advanced) 58](#_Toc349550639)

[Adding Constraints (advanced) 59](#_Toc349550640)

# Introduction

The Broadcom Scene Graph (BSG) library is an object-oriented application framework which simplifies the creation of OpenGL ES2 programs.  It implements most of the tedious boilerplate code associated with writing 3D applications, leaving the programmer free to concentrate on its essential features i.e. creating and animating a 3D scene.

In particular BSG:

* Hides the details of window creation and device input of particular platforms.
* Provides a data structure (a scene graph) for describing the arrangement of the 3D scene.
* Supports an effect-file format for specifying how objects should be rendered.
* Includes tools for loading assets such as meshes and textures.
* Implements a wide set of flexible animation classes for creating dynamic scenes.
* Provides implementations of common mathematical tools for 3D graphics (e.g. vectors, matrices and quaternions).
* Supports text rendering both in 2D and 3D.
* Supports video-texturing.
* Manages objects through reference counting simplifying application memory management.
* Has a general-purpose high-level GUI carousel object.

This document gives an overview of how to use the BSG to create your application. It should be read in combination with the doxygen auto-generated API documentation and with the example applications that ship with BSG. The 3D backgrounder gives an account of some of the basic concepts needed to use BSG, but it is a large topic and some background reading on OpenGL ES or related APIs is recommended.

## Namespaces and Naming

All the names in the BSG live in the “bsg” namespace. To avoid having to prefix everything with BSG and STL identifiers, the code herein assumes:

using namespace std;

using namespace bsg;

BSG class and method names look like this: “ThisIsMyName”. Accessor functions are named like this “GetSomething” and “SetSomething” and other methods have names like “VerbNounPhrase”.

Enumerations are styled “eAnEnum” and its enumerants are like “eAN\_ENUMERANT”.

## Terminology

*Application:* refers to the program running over the BSG. In the framework, application code implements a class derived from the BSG application abstract base class.

*Platform:* refers to the combination of hardware and software the application will be running on. In BSG, the details of each platform are hidden from the application by a platform specific software layer.

*Scene Graph:* refers to a data-structure built by the application which describes a 3D scene. Each node in the graph may contain child nodes, geometry and a camera. Each node also has its own transformation which positions it within the 3D world. The transformation is inherited by all its children, and hence a hierarchy of transformations can easily be constructed. BSG also implements a system of constraints which can be used to override the normal hierarchy.

*Transformation:* refers to a combination of translations (movements), rotations and scales. Each node in the scene graph has a transformation.

*Geometry:* in BSG refers to the combination of a *surface* (a drawable object), and a *material* which is an instantiation of an *effect*.

*Animation:* refers to the process whereby objects in the 3D world are manipulated over time. BSG has a rich set of animation classes which can be attached to any animatable target. The targets need not be part of the scene graph, although they often are.

## 3D Backgrounder

This document is not intended to be a treatise on 3D graphics. There are many texts available which cover the topic is great detail. However, in this section, we do present the bare-bones of the background needed to use the BSG effectively. The terms and concepts introduced here should provide the signposts needed for a more detailed study of the literature.

### Coordinate Systems

Describing a scene in 3D requires us to be able to position, orient and size objects in space. We also need to be able to describe how to view those objects. In common with most 3D systems, BSG uses a 3D Cartesian coordinate system. A position is expressed as a triple (x, y, z) where x is the horizontal position (positive to the right), y is the position vertically (positive up) and z is the position towards or away (positive towards). This system is conventionally used in mathematics and physics and is termed “right-handed” (associating positive x with your thumb, positive y with your index (first) finger, and positive z with your middle finger will show why).

In 3D graphics you will sometimes also encounter positions described as a 4-tuple (x, y, z, w). This is termed a homogeneous coordinate, and is understood to correspond to the 3D coordinate (x/w, y/w, z/w), thus w is effectively a scale factor. Homogeneous coordinates are used to facilitate transformations and in particular are useful in the implementation of perspective projections.

A coordinate (x, y, z) can also be interpreted as a direction or vector. It is the direction of the line from the origin (0, 0, 0) through the point (x, y, z). In terms of homogeneous coordinates (x, y, z, 1) can be interpreted as a position (x, y, z, 0) can be interpreted as a vector (which has no position).

To help applications manipulate objects in 3D, BSG provides the types Vec2, Vec3 and Vec4.

### Transformations

Objects in the 3D world may be moved, rotate and scaled. These kinds of operations are collectively known as transformations. Transformations are the mainstay of 3D graphics. They can be described mathematically via matrices. The BSG has matrix types Mat2, Mat3 and Mat4 which are used heavily internally to calculate and combine transformations. Applications do not often need to manipulate these matrices directly in their C/C++ code, but it is important to know how they work in order to be able to write the shader sections of Effects (described later).

### Color

OpenGL (and hence BSG) has a simple model of color. Colors are described in terms of their three primary components (red, green and blue) and a fourth component which describes the opacity (alpha). Each component conventionally ranges from 0 (fully off) to 1 (fully on). A color (r, g, b, a) can be represented in BSG using the Vec4 class.

Colors in OpenGL correspond roughly to the colors on a computer or TV display and do not correspond very well to the real physical properties of light. It is possible to work with more accurate light models e.g. to use high dynamic range colors in OpenGL ES shader programs (described below), but the final output color channels will always be clamped into the range zero to one.

### Primitives

In a 3D graphics system, objects are described in terms of rendering primitives. The OpenGL family of APIs uses primitives such as points, lines and triangles. Points can be thought of as 2D screen-aligned squares. They can be textured and can be effective when rendering e.g. text or 2D sprites for games. Lines are used less often since their thickness is not modulated by distance. However, they can be useful if the application is essentially 2D, or if the lines are used as markers or indicators in e.g. CAD applications. Triangles are the dominant primitive. Triangles have a facing and can be culled if they are back-facing when rendered.

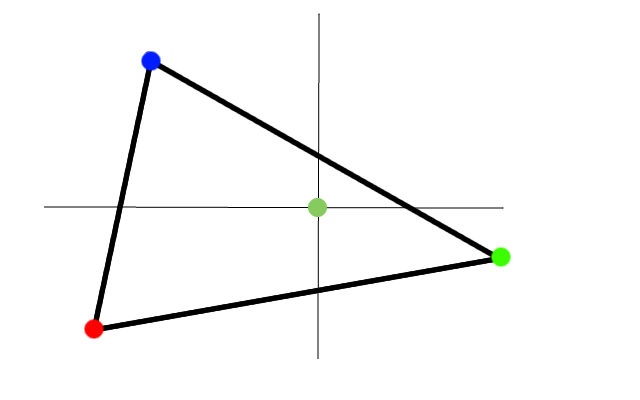
The vertices of primitives are described using 3D coordinates. OpenGL also allows for other *attributes* to be attached to the vertices. The meaning of these attributes is determined by the application, but conventionally they might include texture coordinates, surface normals (vectors perpendicular to the surface used for lighting calculations) and colors.

### Rasterisation

The job of the 3D hardware is to convert the continuous primitives described by the application into discrete pixels. In OpenGL, the notion of a pixel is expanded somewhat to include a position on the display, a color and a depth (and possibly a stencil value). Sometimes the term *fragment* is used to describe this kind of fat pixel.

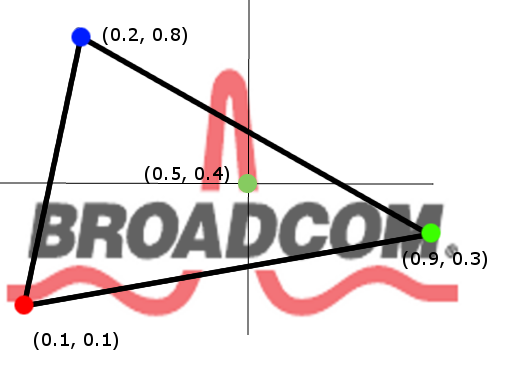
Rasterisation is a form of digital sampling, and as such can be subject to aliasing problems. This is usually manifested as abrupt steps, sometimes called *jaggies*, in near vertical or horizontal edges.

The rasteriser is also responsible for interpolating vertex attributes across the primitives so that they can be used to create the colors internal to the triangle. The following diagram shows a simple case where a color attribute is interpolated:



Thus, the pixel sampled at the cross-hair has a blend of the colors from the triangle vertices weighted according to its proximity to the vertex positions.

The *texture coordinate* is another important type of attribute. Rather than containing a color, texture coordinates describe the position within an image-map from which a color should be sampled. In OpenGL ES, texture coordinates normally lie in the range zero to one. Texture coordinates are interpolated in the same way as colors. This means that each pixel is associated with an interpolated texture coordinate and will fetch data from a different part of the image map.



The red vertex has texture coordinate (0.1, 0.1), the green (0.9, 0.3) and the blue (0.2, 0.8). The pixel at the cross-hairs will be a blend of these values, say (0.5, 0.4), so the color generated from the texture, if point sampling, will be red (coming from the red pulse). For higher quality results, OpenGL ES also allows for bilinear sampling of textures. In the case of bilinear sampling, four colors are read in and blended for every pixel giving a smoother result.

Because 3D objects are subject to arbitrary scaling and rotation, it may be the case that pixels on the screen do not correspond in terms of scale well with the pixels (texels) in the texture. To address this problem, OpenGL ES implements a scheme called mip-mapping. Rather than submit a single texture, applications can submit a set of textures at different sizes (each version is a half the width and height of the previous version and downsampled). The OpenGL ES engine will use the size of the pixel in texture space to select the version of the texture that best matches with the pixels on the display. So a given triangle when projected to a large screen area will use a large texture, but if projected to a small screen area will use a smaller version of the same texture. Mipmaps can also be automatically generated when the textures are created.

Note also, that it is possible to apply several textures to a single triangle. For example, in games, objects might use one texture for a base color and another to model the illumination/shadow on that object.

### OpenGL ES 2.0 Pipeline

The steps going from the application program to the display can be thought of as a pipeline of stages. Real implementations may implement these stages in different ways, but their behavior should not be distinguishable from the simple model presented below.

Display

Rasterisation

Fragment Processing

Vertex Processing

Application

1. The application makes calls to the OpenGL ES API to configure its state and to submit drawing commands. In BSG, most of this work is managed through traversal of the scene graph. State is specified in the effect files and also associated with primitive geometric objects.
2. Vertex data is passed to the vertex processing stage. In terms of a BSG application, the vertex data is passed to a vertex shader (a small program) which inputs vertex attributes and outputs a homogeneous position and (possibly modified or generated) attributes called varyings. The vertex shader is part of the effect file.
3. The varying data passes to the rasteriser where individual pixels are generated along with interpolated varyings.
4. The varyings are passed to the fragment processing stage where the fragment shader (another small program) is run. The fragment shader inputs varyings and converts them to a fragment color. This might involve general calculations and the sampling of textures. Fragment shaders are also part of the BSG effect file.
5. The output fragment is subject to various tests such as depth and stencil tests. In the depth test, each pixel is subject to a test to see whether it is nearer or further away than previous pixels at the same location. If it is nearer it is drawn. If it is further away it is rejected. In OpenGL the function used in the test can be changed. Effective use of depth testing can make applications much faster. Broadcom’s hardware implements a technique called “early-depth testing”. If it can be determined that a fragment is going to fail the depth test early on in the pipeline, then the fragment shader is skipped entirely. This reduces the work done per pixel significantly.
6. Output colors can be blended with any existing colors already in the display buffer according to a flexible scheme of blend equations and functions. These are also specified as part of the BSG effect file.

Both vertex and fragment shaders can be controlled via uniform variables. Uniforms are constant across a draw call i.e. they do not change while rendering a single object. Uniforms can be used to send simple data such as a color or more complex data such as a transformation matrix.

The BSG automatically calculates (lazily) some commonly used uniform values such as transformation matrices.

# Anatomy of an Application

Most BSG applications will have a main routine that looks something like this:

int main(int argc, char \*\*argv)

{

try

{

ApplicationOptions options;

options.SetDisplayDimensions(1280, 720);

if (!options.ParseCommandLine(argc, argv))

BSG\_THROW(“Command line not recognized”);

Platform platform(options);

Planets app(platform);

return platform.Exec();

}

catch (const Exception &e)

{

cerr << "Exception : " << e.Message() << "\n";

}

}

## Application Options

The ApplicationOptions class contains settings used to configure the application. They can be set programmatically by the application as above:

options.SetDisplayDimensions(1280, 720);

or they can read from the command-line with:

options.ParseCommandLine(argc, argv);

The method returns true if the command-line arguments are successfully handled. In the sample code above, we have requested a 1280x720 display size which could be overridden by the command-line argument “d=1920x1080”, for example.

The application options class is used to control how the platform is initialized. It includes:

* Resource folder location.
* Display/window size and position.
* Display color, depth and stencil depth.
* Multisample flag.
* Swap interval.
* Stereo flag.
* Stretch to fit display flag.
* Color buffer preserve flag (not recommended due to its performance impact).
* Render to pixmap flag (not recommended due to its performance impact).
* Switch to control the diagnostic HUD.
* Rate-multiplier setting for control over animation speed. By default this is set to real-time.

Applications wishing to parse their own specific command-line arguments can extend the command-line parser by supplying an argument parser derived from ArgumentParser. The class should provide a parsing method to recognize and process the new arguments and a usage method to print help information associated with them.

Note that applications are not required to use the BSG command-line argument parser and can handle things their own way if necessary. However, an ApplicationOptions object is needed to construct the platform, so applications will need to use this object to configure the platform at start-up.

## Platform

BSG can run over multiple platforms. The details of how the system is initialized are hidden inside the Platform class which is implemented specifically for each target platform. Applications do not need to worry about these differences.

*Important note:* the platform object is responsible for creating the display surfaces and GL context. Do not attempt to call any GL or EGL functions before the platform has been created. Some BSG objects call the GL API, so you should also avoid creating these objects before the platform has been initialized. It is recommended that your application object manage BSG objects, because an application object cannot be created without a platform, so by construction a platform must exist at this point.

## Application

The core of a BSG application is the Application class. The BSG uses the interface of this class to control the behavior of the application. Each specific application must derive its own class from the BSG application class and supply a set of virtual methods to specialize its behavior. These are:

virtual void RenderFrame(); // Required

virtual bool UpdateFrame(int32\_t \*idleTimeMs); // Required

virtual void KeyEventHandler(KeyEvents &); // Optional

virtual void MouseEventHandler(MouseEvents &); // Optional

virtual void ResizeHandler(uint32\_t width, uint32\_t height); // Optional

The framework follows this sequence:

KeyEventHandler(events);

MouseEventHandler(events);

if (UpdateFrame(&idleTimeMs))

RenderFrame();

The KeyEventHandler() is called to allow the application to handle USB keyboard (or remote control) key events. MouseEventhandler() is called to process any events from a connected USB mouse.

Next, the BSG framework calls UpdateFrame(). This is the application’s opportunity to update its scene and to notify the framework if any changes have been made. If UpdateFrame() returns false, then the BSG framework will not trigger a render and will wait for idleTimeMs before doing another update. This can be useful for power and CPU cycle saving in GUI applications which display a static menu for example.

In RenderFrame(), the application should perform all its rendering, typically by using the BSG scene-graph machinery (see below). It should *not* call eglSwapBuffers() as this is handled automatically by the BSG framework. A typical RenderFrame() might look like this:

void MyApp::RenderFrame()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

// Draw my scene

RenderSceneGraph(m\_root);

}

The BSG sets the viewport automatically prior to calling RenderFrame(). You can override this using Application::glViewport() if required (note that the global ::glViewport() should be avoided as this will not work correctly in stereoscopic applications).

The ResizeHandler() is called whenever the display window is resized. On some platforms this may never happen. It provides an opportunity for the application to resize objects and fonts as necessary. Applications should also notify their cameras of the screen size change.

void MyApp::ResizeHandler(uint32\_t width, uint32\_t height)

{

m\_camera->SetAspectRatio(width, height);

}

# Introducing the Scene Graph

BSG uses a scene graph to describe a 3D scene. A scene graph comprises a set of nodes linked to one another in a parent-child relationship. The structure of the graph describes the geometric relationships between its component parts. A scene graph must not contain cycles but nodes can be shared. A graph might look like this, for example:

Every graph will have a root object. Everything in the scene will depend, either directly or indirectly on the root node. Thus an application can manipulate the entire scene by modifying the root node. In the graph above, nodes A and B are directly linked to the root. This means that they will be subject to all the transformations specified in the root, as well as their own local transformations.

For example, if the Root contains a transformation “move left by 20”, and node A contains the transformation “move right by 5”, then the geometry held in node A will be moved left by 15. If node B contains “move left by 5”, then its geometry will be moved left by 25.

Node C is interesting because it is a child of both A and B. When the tree is processed by the BSG, node C will be visited twice, once from A and once from B. Any objects in C will be rendered twice. If node C contains a transformation “move up by 10”, then one instance of the node will be “moved left by 15 and up by 10” (via A) and the other will be “moved left by 25 and up by 10” (via B).

Sharing nodes provides a way to replicate geometry without having to replicate the graph that describes it. An alternative would be to create two nodes C and D where D is replica of C. Whether to share or replicate a node depends on whether the application ever needs to be able to control the instances independently of one another.

As well as child nodes, nodes can also contain geometry and a camera. These will both inherit the accumulated transform for their node. The geometry describes the drawn objects in the scene and the camera describes how the scene should be projected to a display. Because the cameras are part of the scene graph, they can be animated to move along with geometry, for example. Multiple cameras can be inserted into the graph. The application can choose which one to use when the scene is rendered.

## Inside Scene Nodes

Each scene node can contain a number of geometry items and optionally a camera item.

Geometry

Camera

Geometry

* Camera objects can be placed at any scene node. This means that cameras are first class scene graph objects and can be animated. Being part of the graph means that they can be made to follow animating geometry for example.

By default, the rendering engine will use the first camera it finds. This is useful behavior if there is only one camera. If there is more than one camera, then the application should specify which camera is to be used when rendering the scene graph.

* Geometry objects combine (lists of) Surfaces and Materials. Think of a geometry object as clothing the surfaces with their materials. So if the surface is a sphere, and the material applies an earth texture, then the geometry is a model of the earth.
* Surfaces contain the actual triangles, points or lines to be drawn. They can be created by loading geometry files or by using one of the predefined shape-factory classes.
* Materials describe how the surfaces should be rendered. A material is an instantiation of an effect. Its job is to fill in the uniform variables exposed by an Effect. So if the Effect is “metal”, then a material using the effect might be “golden shiny metal” or “silvery matt metal”.
* Effects describe the GL state and shaders (vertex and fragment) which should be used when rendering a surface. Effects are usually loaded from a text file in the BSG effect file format. This is a human-readable format and is described below.

Note that, just like scene nodes, all these objects are potentially shareable. So, for example:

* A material could be applied to multiple surfaces. This is a common example of sharing since it is often the case that the same material should be applied to multiple surfaces.
* An effect might be shared by several materials. For example, an effect file might describe a “plastic” material. Instances of this effect might be “red plastic” or “blue plastic”.
* A surface might be shared by several geometry nodes. This makes the most sense if the geometry nodes are held in different scene nodes since, that way, the surfaces will appear in different 3D positions. However, it is not impossible to have a surface appear multiple times in the same scene node and this might be useful if the material contained additional geometric transformations such as skinning, or a fragment shader that selectively rejects pixels.

The “planets” example builds a scene graph describing the solar system. The following diagram is a simplified version of this graph and shows how it models the dependencies between the various astronomical bodies. Ovals represent scene nodes, triangles cameras and rectangles geometry.

## Example Scene Graph

An application can render this scene graph by simply calling:

RenderSceneGraph(root);

The BSG gives some additional controls over how objects are rendered via the RenderOptions class, so a typical piece of code to render a scene graph might look like:

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

RenderOptions options;

options.SetEnableViewFrustumCull(true);

RenderSceneGraph(root, options);

The options currently support one flag which controls whether view frustum culling is applied. This is a technique that tries to remove objects from the render list if it can be determined that they lie outside of the field of view of the camera. It relies on all the objects in the scene having accurate bounding volumes (which is true for geometry generated using standard BSG functions). However, view frustum culling does imply a run-time overhead and if an application knows that all the objects will always or nearly always be visible, it should not be used.

The RenderSceneGraph() method also has an optional camera argument which should be used if the scene graph contains multiple cameras.

## Notes on Handles

BSG supports application development by maintaining a central repository for its scene-graph objects and granting access to these objects via “handles”. Applications do not normally manipulate objects directly. They must go via the handle. Handles look syntactically like pointers in application code i.e. the “operator->” has been overloaded for handles. Thus e.g. one can write:

handle->Method();

and the result will be to invoke Method() on the object represented by handle. Internally, BSG will convert the handle to the underlying pointer and invoke the appropriate method. This is a lightweight operation. As a convenience for debugging purposes, for debug builds only, the raw pointer is also stored in the handle. Do not use this pointer in your application.

Handles can be in one of two states. Simply declaring a handle will create a “null” handle:

SceneNodeHandle sceneNode; // This handle is “null”

Like a null pointer, using this handle to try to reference a SceneNode will result in undefined (probably a crash) behavior. Handles have an IsNull() method to test whether a handle is initialized. Alternatively, a handle can be created automatically initialized via:

SceneNodeHandle sceneNode(New);

This arrangement means that we can safely and easily create containers such as vectors, lists and arrays of handles. The handles will be initially set to “null” and can be populated as required.

Handles control a reference count on the objects that they reference. When the reference count falls to zero, the referred object is automatically deleted. Handles can be freely copied and assigned without worrying about keeping track of the underlying object. The object will be retained for as long as there are live handles referencing it. Note this also means that if an application created a handle in the heap, either directly or indirectly, it will only delete the underlying object after the heap based handle is deleted.

Handles are small (no bigger than a pointer) so can be stored compactly. However, whenever a handle is copied or assigned, the reference counts held in the underlying objects are adjusted appropriately. For this reason, it is sometimes better to pass handle parameters as constant references. This has the same semantics as pass-by-value, but has better performance. You should only do this if you are passing a handle in a transient way however, since the reference count will not be updated. If you are storing a copy of the handle in an object, you should do so by value and not by reference otherwise you may find that the original handle deletes the underlying object, leaving you with a reference to an invalid handle. If you have a by-value copy of the handle, the underlying object cannot be deleted behind your back.

In summary, never *store* a handle by reference or pointer, but if you are passing them as arguments it can be more efficient to pass by reference.

Handles can also be associated with a name:

SceneNodeHandle sceneNode(New, “name”);

Names, where used, must be unique. A previously named handle can be retrieved via:

SceneNodeHandle sceneNode(“name”);

If the name is not found, then a “null” handle will be created instead.

Handles can also be associated with an integer ID. Unlike names, the ID of a handle is not required to be unique. By default, all handles are created with a zero ID.

SceneNodeHandle sceneNode1(New, 23);

SceneNodeHandle sceneNode2(New, “fred”, 23);

There are methods on the handles to retrieve the name, id and underlying pointer. The pointer should be used with care. If the handle to a particular pointer goes out of scope, then the pointer will become invalid as the memory will be returned to the system. It is acceptable to use these pointers only for transient purposes.

# Hello BSG

The set of examples in “bsg\_apps” demonstrate how to use the BSG to create a variety of different effects. In this section, we describe how to build applications, what they do, and how the “hello\_cube” set of examples work in detail.

## Building BSG under BSEAV

The BSG build system makes all the components required to compile and run a BSG application and installs them in the binary (NEXUS\_BIN\_DIR ) folder. This folder should be mounted on the board in a convenient location.

Run the platform script for your board on the build-machine before starting any builds.

### Building Stand-alone

To build an application stand-alone e.g. as a release, user-mode, single process application:

export B\_REFSW\_DEBUG=n

unset NEXUS\_MODE

unset CLIENT

cd BSEAV/lib/framework3d/bsg\_apps/hello\_cube

make

This will build:

* Nexus library and Linux kernel module
* OpenGL ES driver
* OpenGL ES platform layer
* Third party libraries libpng, libz and libfreetype
* The BSG core library
* The application

You will need to install the Broadcom kernel module and set your library path e.g. for a Nexus user-mode build (where NEXUS\_BIN\_DIR points at your Nexus binary folder):

insmod $(NEXUS\_BIN\_DIR)/bcmdriver.ko

export LD\_LIBRARY\_PATH=$(NEXUS\_BIN\_DIR)

If you do not reboot the board and do not change to a different shell, you should not need to repeat these commands.

To run the application on the board, go into the target “bin” directory and execute the application i.e.

cd $(NEXUS\_BIN\_DIR)/bsg/hello\_cube1

./hello\_cube1 d=1920x1080 +m

To run an application in a different location, you can use the “res=” argument to specify where to look for the application resources e.g.

hello\_cube1 d=1920x1080 res=bsg/hello\_cube1/resources

### Batch Building and Cleaning

To build all the BSG applications, set your Nexus build environment variables and:

cd BSEAV/lib/framework3d/bsg\_apps

make

To clean the BSG build:

cd BSEAV/lib/framework3d/bsg\_apps

make clean

This will remove all the code and resources installed to nexus/bin and all the intermediate files.

### Using the BSEAV build system

To build BSG as part of the BSEAV build:

cd BSEAV/build

make bsg

This will build and install all the BSG applications to BSEAV/install/bsg. To clean up do:

cd BSEAV/build

make bsg\_clean

## Building BSG for Trellis

The Trellis platform for BSG is currently under development.

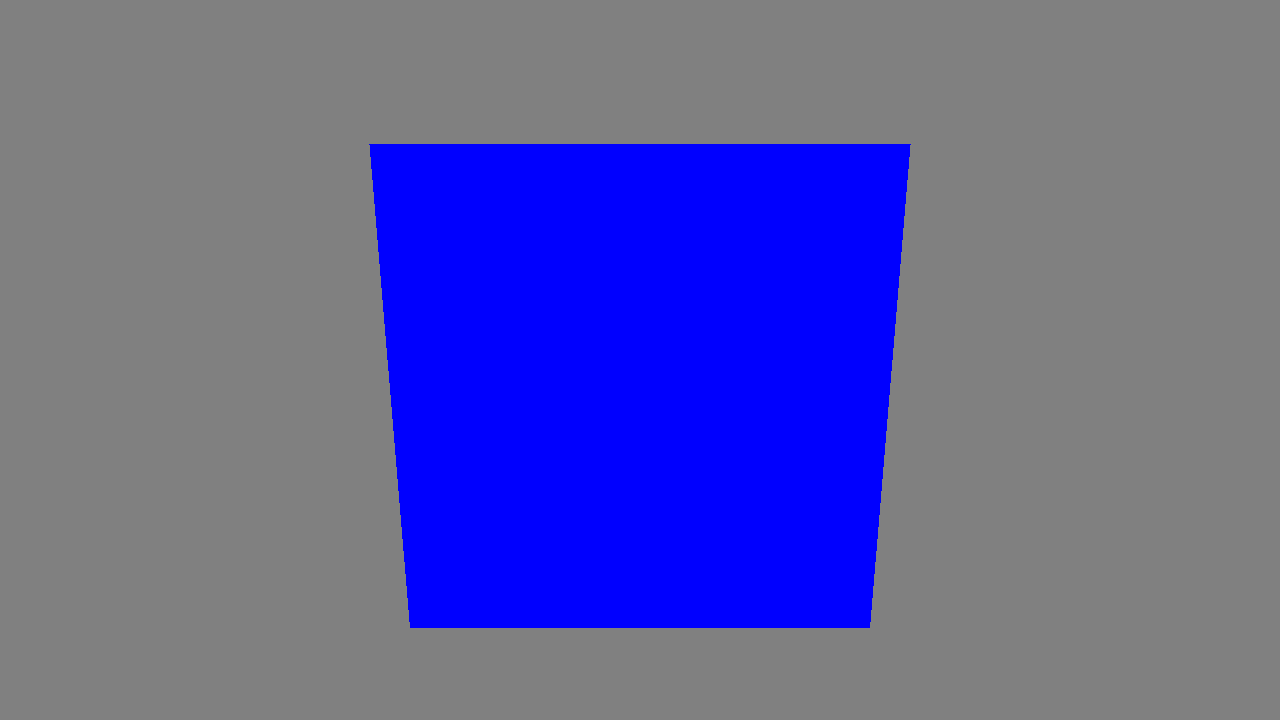
## API Documentation

The BSG API is documented using Doxygen. To generate the documentation from the source code, go to the BSG core source folder and enter:

make –f bsg.mk docs

The documents are copied into the framework3d/doc folder.

## Hello Cube 1 – Basic Setup



The constructor for the application looks like this:

HelloCubeApp1::HelloCubeApp1(Platform &platform) :

Application(platform),

m\_root (New),

m\_camera(New)

{

// Setup a basic camera

m\_camera->SetClippingPlanes(0.5f, 100.0f);

m\_camera->SetYFov(65.0f);

// Add camera into its node and position it

SceneNodeHandle cameraNode(New);

cameraNode->SetCamera(m\_camera);

cameraNode->SetTransform(

CameraTransformHelper::Lookat(Vec3(0.0f, 1.5f, 8.0f), // Where

Vec3(0.0f, 0.0f, 0.0f), // Lookat

Vec3(0.0f, 1.0f, 0.0f))); // Up-vector

// Load the effect file for the cube and create a material with it

EffectHandle cubeEffect(New);

cubeEffect->Load("cube\_flat.bfx");

MaterialHandle cubeMat(New);

cubeMat->SetEffect(cubeEffect);

// Create the geometry and add to its node

GeometryHandle cubeGeom(CuboidFactory(5.0f).MakeGeometry(cubeMat));

SceneNodeHandle cubeNode(New);

cubeNode->AppendGeometry(cubeGeom);

// Construct the scene graph (root with a camera and cube child)

m\_root->AppendChild(cameraNode);

m\_root->AppendChild(cubeNode);

// Init any global GL state

glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

}

This is about the minimum that is required to create a useful application. First we create the root scene node and camera node. These handles are required elsewhere in the application, so we have chosen to hold them as members of the application object. The scene nodes that hold the camera and cube are created locally since they are only referenced here. Note that these handles will be inserted into the scene graph and will therefore be retained for as long as the root node persists.

Next we initialize the camera and set its transformation. We then load in an effect (“cube\_flat.bfx” ) which implements simple flat-shading. We bind a material using the effect. Note that this particular effect does not expose any uniform variables so there is nothing more to do to create the material.

The geometry of the cube is created using the CuboidFactory utility class. This generates the underlying surface and binds it to the cube material. See the “geom\_viewer” example to see more geometry types.

Finally, we connect everything together. Both the camera and cube nodes are parented to the root node.

The RenderFrame() method for this application looks like this:

void HelloCubeApp1::RenderFrame()

{

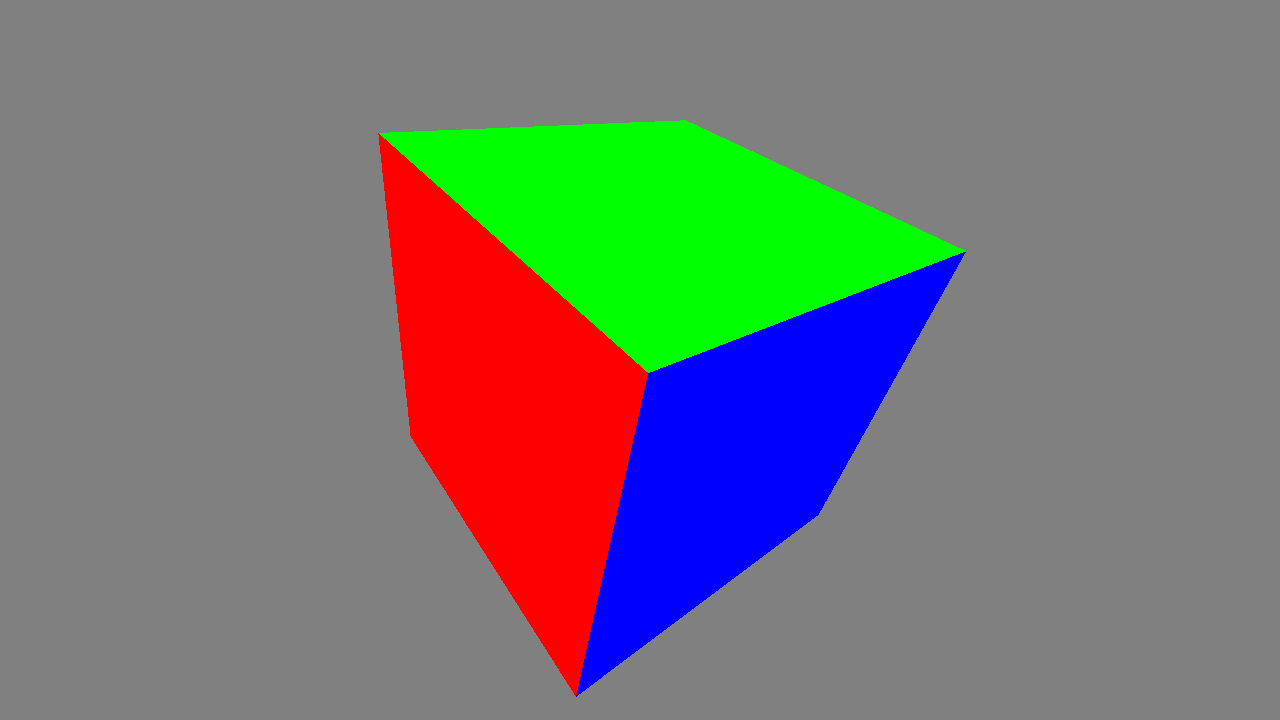
glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

RenderSceneGraph(m\_root);

}

Running this application should result in displaying a blue quadrilateral, which is a view of the cube. You can experiment with changing the transformation of the cube to get different orientations of the cube, or you can change the camera parameters to modify the view.

## Hello Cube 2 – Moving Stuff About



The first hello cube application is not very interesting as the cube does not move. In the next application, we show how to add some animation. Animations are ephemeral objects and are not considered part of the scene graph. They are not handles and are typically created using the “new” operator and are automatically deleted on completion (auto-deletion can be disabled if required).

The job of an animation is to modify data over time. For example, an animation might be used to control a position or rotation within a scene graph. They can be used to control virtually any data whether it’s part of a graph or not. Animations can be finite (e.g. move from A to B in 10 seconds) or infinite (e.g. follow a circular path forever).

In the second cube example, we have an example of an infinite rotation which is attached to the cube’s scene node.

The new code adds the following lines:

Time now = FrameTimestamp();

AnimBindingLerpQuaternionAngle \*anim =

new AnimBindingLerpQuaternionAngle(&cubeNode->GetRotation());

anim->Interpolator()->Init(now, now + 10.0f, BaseInterpolator::eREPEAT);

anim->Evaluator()->Init(Vec3(0.7f, 1.0f, 0.0f), 0.0f, 360.0f);

m\_animList.Append(anim);

The FrameTimeStamp() method obtains the time for the current frame. Note that this method will return the same time until a new frame begins which is helpful when setting up multiple animations in different parts of an application which need the same reference time.

Animations are built in three stages. First the animation object must be created. The type of the animation will determine what its properties are and what kind of targets it can be connected to. In this case, we are creating a quaternion-angle interpolator which should be attached to a quaternion (quaternions are BSGs way of describing rotations – applications do not need to worry about their details). So, here we attach it to (a pointer to) the rotation part of the transformation in the cube-node. Note that animations should be allocated via “new” giving their target as the constructor argument. The BSG will delete animations automatically when they complete. An application can optionally install a notifier object which will be called when an animation completes.

Animations comprise two components – an interpolator and an evaluator. The interpolator generates an interpolated floating-point value depending on its start and end time. The interpolated value is fed into the evaluator. This then maps the interpolated float into the target type. For example, it might use the interpolant as the input to a circle generator or a spline evaluation. The most common evaluators simply use the interpolant as the input to a suitable interpolation between start and end values.

In the example, we are using a linear interpolator targeting a quaternion. The evaluator for quaternions is mapped to a spherical linear interpolation which gives a natural interpolation behavior for rotations. The quaternion is specified to be along the axis (0.7, 1.0, 0.0) and has angle value 0.0 at time “now” and 360.0 at time “now + 10”. This causes the cube to rotate once every ten seconds. As the interpolator is specified to repeat, the rotation will continue indefinitely. For animations that continue indefinitely, the “eREPEAT” and “eMIRROR” variants are very useful, since they use a modulo of the extrapolated value. This prevents the values becoming huge and potentially overflowing, or becoming inaccurate over long periods of time, as could happen with “eEXTRAPOLATE”. For finite animations, the “eLIMIT” mode should be used. This will clamp the evaluator to zero before its start time and one after its end time.

Applications with animations should extend the UpdateFrame() method to invoke the animators on their moving objects. This is as simple as:

bool HelloCubeApp2::UpdateFrame(int32\_t \* /\*idleMs\*/)

{

m\_animList.UpdateTime(FrameTimestamp());

return true;

}

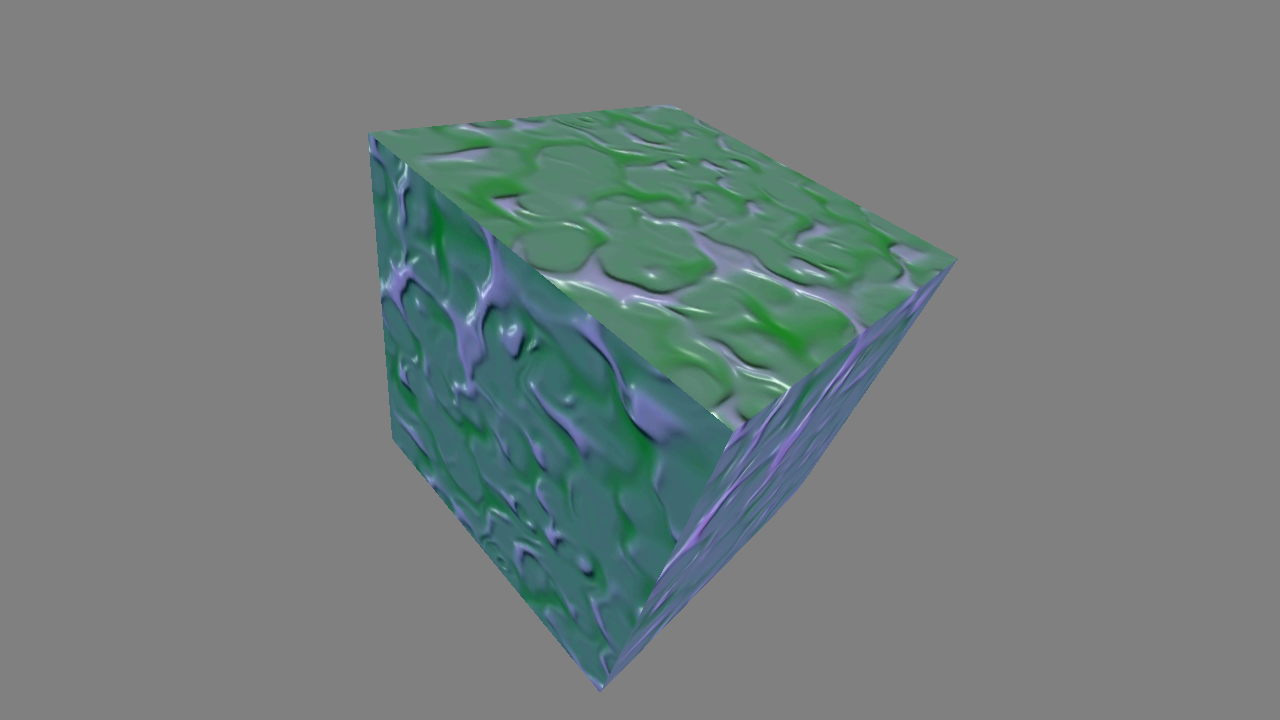
The “true” return value indicates that something has changed and that the frame should be re-rendered. The UpdateTime() method returns a Boolean indicating whether any animations are active, so this can be used as the return value of UpdateFrame() if desired.

Applications can create and manage several animation lists. This might be useful to partition animations into different kinds which might be serviced at different rates, for example.

By using the FrameTimestamp() method, we can ensure that the animations proceed in real-time regardless of the actual display frame rate. This helps keep applications looking smooth. However, there is no requirement that the real time be used. An application could increment its own counter for each frame and use this as the time variable. Note, however, that animations are not designed to be wound backwards.

Applications which are idle (e.g. a static menu) should return false from UpdateFrame() and set the ideMs argument to a suitable idle time. This will prevent any rendering taking place for that time and will save power and computational costs.

## Hello Cube 3 – Adding Texture



In the next hello-cube application, we add a texture to the surface of the cube. To put a texture into an object, we need to do several things:

1. Load an effect file which uses a texture.
2. Load an image file containing the texture data.
3. Use the image to create a new texture handle.
4. Bind a material to the effect and connect the texture object to the material.

Effects that use textures must include some vertex shader code to generate texture coordinates (either from attributes, or otherwise), and must use these coordinates to read from a texture sampler in the fragment shader. The sampler must also be declared in the effect file.

Creating the effect and material works the same as before (except we use a different effect file):

EffectHandle cubeEffect(New);

cubeEffect->Load("cube\_tex.bfx");

MaterialHandle cubeMat(New);

cubeMat->SetEffect(cubeEffect);

Loading data for a texture is most simply done using the Image or ImageSet class. The former loads a single texture and the latter a whole mipmap pyramid. The Image classes support the loading of PKM (or ETC1), PNG and RAW files.

TextureHandle cubeTex(New);

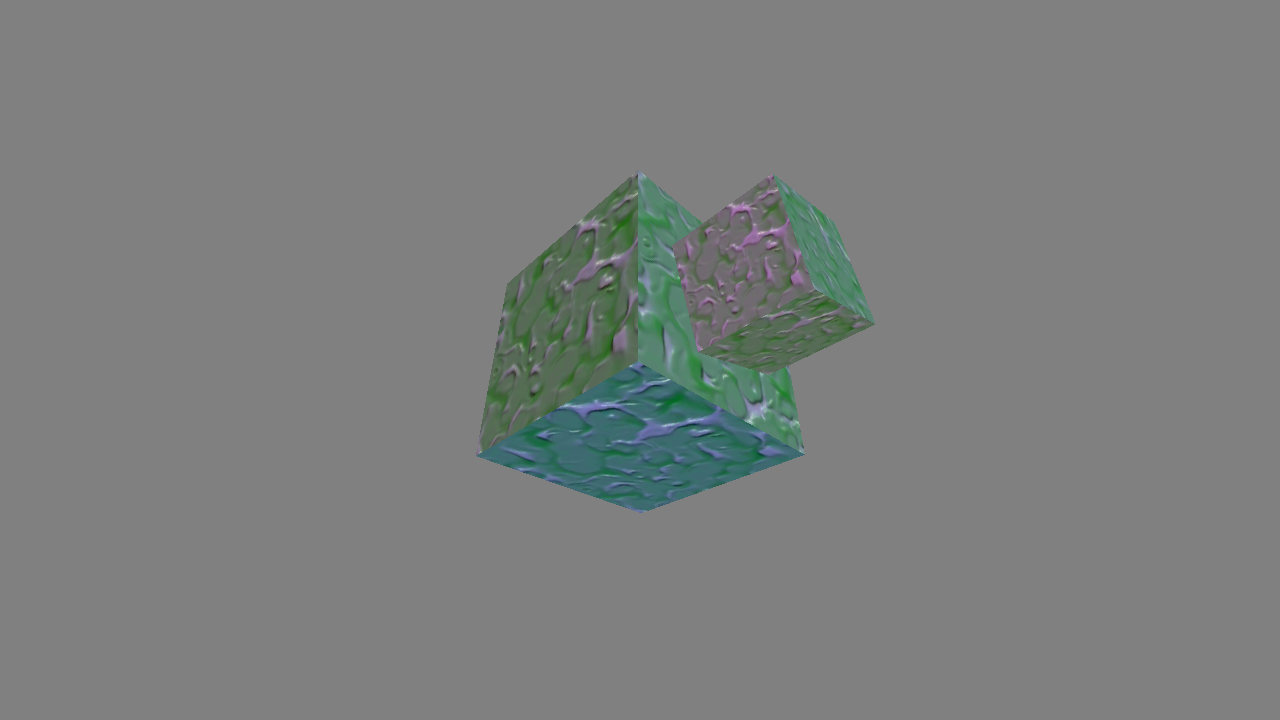
cubeTex->TexImage2D(ImageSet("cube", "pkm", Image::eETC1));

Binding the material to the texture is done via SetTexture(). The name must match the texture sampler declared in the effect file.

cubeMat->SetTexture("u\_tex", cubeTex);

The “geom\_viewer” example shows how texture coordinates are mapped on to each of the standard primitives.

## Hello Cube 4 – Using the Hierarchy



One of the principal goals of a scene graph is to make it easy to construct models in which some parts depend on the positions and orientation of others. This kind of arrangement happens all the time in the real world. For example, a gun on the turret of a tank will inherit the position and orientation of the tank, the rotation of the turret and the elevation of the gun itself. Describing this as a single transformation is complicated, but by decomposing it into its constituent parts, the problem is much more tractable. In BSG, the scene graph manages this complexity and so to model the tank example, we would create a tree of nodes with the tank body at the root, the turret depending on that (offset and rotated). The gun would be a child of the turret and have its own local transformation for elevation.

In the fourth hello-cube example, we create a very simple hierarchal model comprising two cubes. The first cube is the same as our previous example. The second cube depends on the first, being offset and rotated with respect to it.

Creating the hierarchy in BSG is simple. The following new code is added:

// Create the geometry and add to the nodes

GeometryHandle cubeGeom = CuboidFactory(3.0f).MakeGeometry(cubeMat);

SceneNodeHandle cubeNode1(New);

SceneNodeHandle cubeNode2(New);

cubeNode1->AppendGeometry(cubeGeom);

cubeNode2->AppendGeometry(cubeGeom);

// Position and scale node2 to be on top face of cube in node1

cubeNode2->SetPosition(Vec3(0.0f, 2.25f, 0.0f));

cubeNode2->SetScale(Vec3(0.5f, 0.5f, 0.5f));

We now have two cube nodes, which share the cube geometry (since both objects are exactly the same there is no point generating a new piece of geometry). The second node is scaled and offset so as to place it on top of the parent cube. We can then arrange the hierarchy with the root node owning the first cube and the first cube owning the second.

// Construct the scene graph (root with a camera and cube child)

m\_root->AppendChild(cameraNode);

m\_root->AppendChild(cubeNode1);

// Cube2 hangs off cube1 and hence inherits all its transformations

cubeNode1->AppendChild(cubeNode2);

Finally, we can animate the second cube in the same way as we did the first:

AnimBindingLerpQuaternionAngle \*anim2 =

new AnimBindingLerpQuaternionAngle(&cubeNode2->GetRotation());

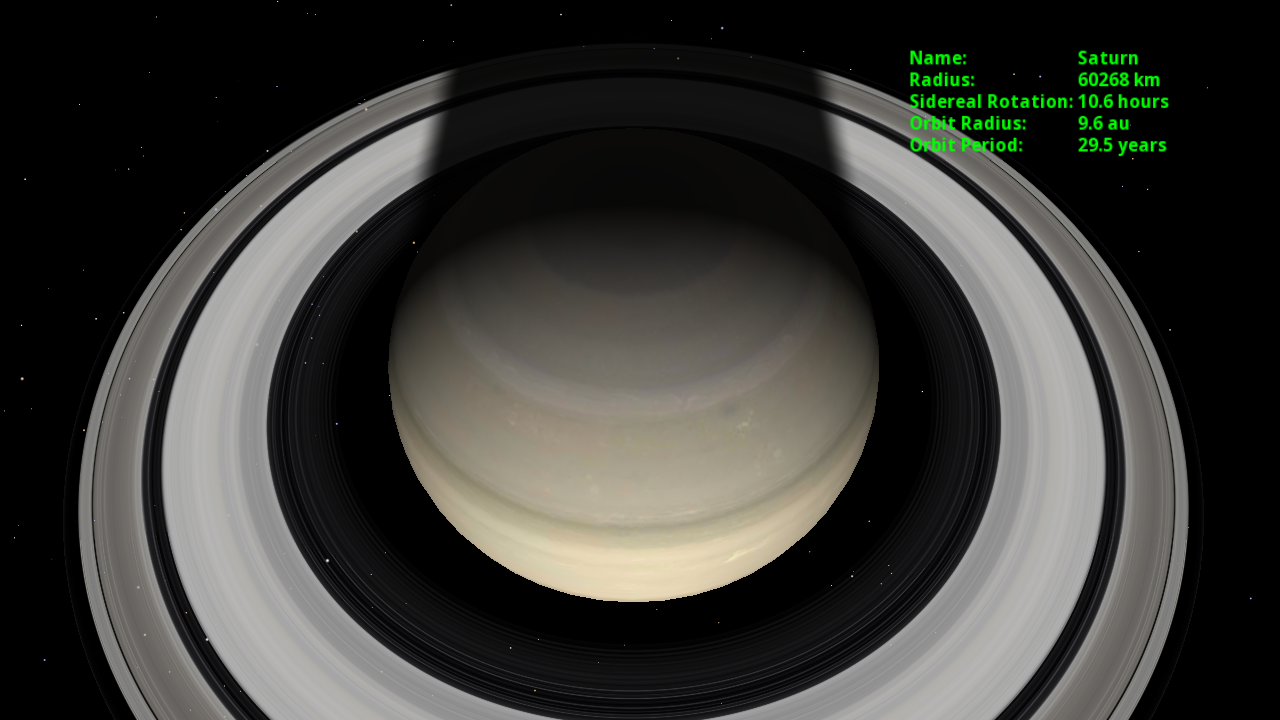
anim2->Interpolator()->Init(now, now + 7.0f, BaseInterpolator::eREPEAT);

anim2->Evaluator()->Init(Vec3(0.0f, 1.0f, 0.0f), 0.0f, 360.0f);

m\_animList.Append(anim2);

## Planets

As described above, the planets example makes extensive use of the scene-graph hierarchy as well as exercising the constraints mechanism for positioning and orienting the camera.



## PG

The program guide application is a fairly complex example of how BSG can be used to create a 2D UI with a number of animation effects. It shows how graphs can be created and destroyed dynamically.



# Textures and Images

In “Hello Cube 3”, we showed how to create, load and bind a texture. BSG splits the process of texturing into two stages. Textures are loaded from a file using the Image classes. Images are usually transient objects which exist just long enough to retrieve the data from a file and pass it on to a texture object. A TextureHandle object is where image data is passed into the OpenGL ES driver. A TextureHandle does not retain the image data passed to it since this would waste memory. The data is simply passed to the OpenGL ES driver which will create its own internal copy.

The Image loaders Image, ImageSet and CubeImages are “virtual constructors” in the sense that they use their arguments to determine what kind of image data to load. This data is then made available to textures via a common interface.

# Geometry

Creating 3d objects is fundamental to creating a 3d application. BSG has a number of “factory” classes which can be used to generate geometry of various common shapes. Applications can use these “off-the-shelf”, can modify their behavior via modifiers or can write their own geometry generation code. Details of the latter are presented in the section “Extending BSG”.

The standard factories are:

* SphereFactory: creates a sphere
* QuadFactory: create and axis aligned quadrilateral.
* CuboidFactory: creates an axis aligned cuboid.
* DiskFactory: creates a circle shape.
* RingFactory : creates a disk with a hole.
* ConeFactory: creates an open ended cone.
* RoundedRectFactory: creates a rectangular shape with rounded corners
* PanelFactory: creates a rectangular shape with a border.

In addition there is:

* ObjFactory: load an Alias/Wavefront “obj” file. Most 3D content creation tools can export to this simple format.

We have already seen how to use the factories in the various “hello\_cube” examples. Each of the standard factories has a MakeGeometry() method that binds the shape to a material. They also have a MakeSurface() method which returns the surface unbound. This can be useful if you wish to populate a single geometry with several surfaces.

Because “obj” files can define multiple surfaces, the ObjFactory only exposes a MakeGeometry() method.

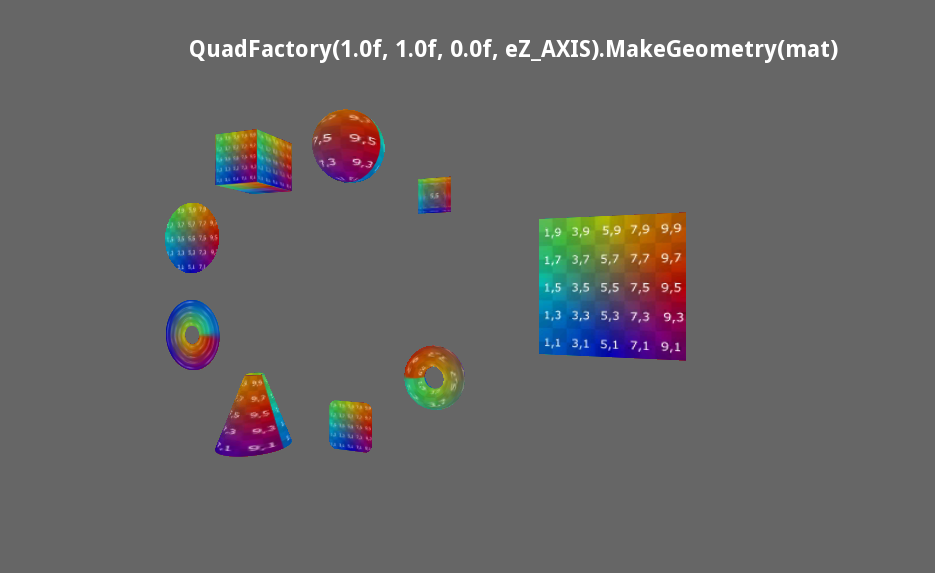
By default, the standard geometry factories generate objects with position, normal and texture coordinate data. These can be individually suppressed to save space. We can slightly improve the “hello\_cube1” example by suppressing the texture coordinate generation for the cube using the No() modifier.

GeometryHandle

cubeGeom(CuboidFactory(5.0f).No(StdFactory::eTEXCOORD1).MakeGeometry(cubeMat));

In practice this will make very little difference to this particular example, but when the geometry being created is large, it is worth doing.

The “geom\_viewer” example demonstrates the various types of geometry available and shows how the texture coordinates have been mapped onto the different primitive shapes.



The various factory classes can be used as examples of how to generate geometry and can be adapted to create other forms.

# Animation Details

The animation framework in BSG helps with creating moving scenes with real-time behavior. Usually animations are connected to scene-graph objects, but this is not required. The animation framework can be used to control any suitable variable including application variables.

In an animation, there are two actors: the animator and the target. In the BSG framework, the animators are derived from an AnimBinding<E, I>, where “E” is the type of the evaluator (see Cube 2 above) and “I” is the type of the interpolator. An animation binding can be connected to an animation target provided that the output from the evaluator can be sensibly converted to the type of the target (given by E::Type).

The framework exports a predefined set of common animation binding classes with the naming convention AnimBindingInterpolatorEvaluator where the available interpolators are:

* Lerp, Hermite, DoubleHermite, CubicIn, CubicOut, Cubic, CircIn,   
  CircOut, Circ, OvershootIn, OvershootOut, Overshoot, ElasticOut

The evaluators are:

* Transform, Float, Vec2, Vec3, Vec4, Quaternion, QuaternionAngle, TextParams, CatmullRom, Circle, Time

All animation bindings derive from the base class AnimBindingBase, so they can be treated polymorphically. The container AnimationList is used to hold a collection of animations.

To be the target of an animation, a variable must be derived from the template base class AnimTarget<T>. The template parameter describes the type of values that can be set into the target.

For many cases, the animation target will derive directly from the target type. BSG provides a template class Animatable<T> to implement this behavior.

template <class T>

class Animatable : public T, public AnimTarget<T> { ... }

So, for example, an Animatable<Vec3> *is* a Vec3 which happens to also be animatable and can be used as the target of an animation binding. For built-in data types like “float” (which are not classes and hence cannot be a base class), the BSG provides wrapper classes like Float (actually a Number<float>), so we can define an Animatable<Float>.

BSG provides type synonyms for the common geometric and numeric types:

* AnimatableVec2, AnimatableVec3, AnimatableVec4, AnimatableIVec2, AnimatableIVec3, AnimatableIVec4, AnimatableQuaternion, AnimatableTime

Sometimes, for example when setting a light position in multiple shaders, it is useful to be able to control several targets with a single animatable object. BSG provides:

template <class T> class AnimatableGroup : public AnimTarget<T> { ... }

for this purpose. When an AnimatableGroup is animated, it fans-out the changes to all its children. All the targets will be set to the same value. Reading from an AnimatableGroup will return the most recently set value, whether there are any targets in the group or not.

Uniforms can also be animated and the BSG provides the following animatable targets:

* AnimatableUniform1f, AnimatableUniform2f, AnimatableUniform3f, AnimatableUniform4f
* AnimatableUniform1i, AnimatableUniform2i, AnimatableUniform3i, AnimatableUniform4i

For example, an AnimatableUniform1f is (ultimately) an AnimTarget<float> and an AnimatableUnform2f is an AnimTarget<Vec2>. References to these types are returned by the material GetUniform1f(), GetUniform2f() etc. So to bind an animation to a material, an application can say e.g.

AnimBindingLerpFloat

\*floatAnim = new AnimBindingLerpFloat(&myMaterial->GetUniform1f(“u\_myFloat”));

For this to work correctly, the material must have a uniform variable called “u\_myFloat” defined in its associated effect.

## Extrapolation

When creating an animation you can specify the behavior of the interpolators for values outside of the start-end time range. For example:

AnimBindingLerpFloat \*rot = new AnimBindingLerpFloat(&mat->GetUniform1f("u\_rot"));

rot->Interpolator()->Init(now, now + duration, **BaseInterpolator::eREPEAT**);

rot->Evaluator()->Init(0.0f, 1.0f);

The options are:

* eLIMIT – which clamps the interpolator result to zero before the start, and to one after the end. This is the default behavior.
* eREPEAT – which performs a modulo operation repeating the function of the interpolator indefinitely (both before and after the begin and end times). When used with positions, this may produce discontinuities, but it is very suitable for angles.
* eMIRROR – which flips the sense of the interpolation every other span and extends the pattern indefinitely. Mirror is good for oscillating movements.
* eEXTRAPOLATE – which simply continues the function outside of the range.

These modes are depicted graphically for a linear interpolation below:

See the sample application “hello\_anim” to see these effects in action.

# Going Deeper into the Scene Graph

For many applications, the basic scene graph machinery is adequate. However, sometimes it is necessary to work with more complex relationships or to modify the behavior of the renderer to achieve special effects.

A lot of the mechanisms discussed in this section are used in the “planets” example. It is worth looking at this application to see these features in action.

## Constraints

The standard scene graph describes the position/orientation of a child node as the combination of the parent’s transformation and the local transformation of the child. For most purposes this is a natural match for the behavior of real-world objects. However, consider the example of a camera which is tracking an object. The camera might be positioned according to one transformation, but its “look” direction is determined by a second transformation (i.e. the position of the target node).

To facilitate cases like these, the BSG implements a system of constraints. A constrained node is one that depends on more than one parent node for its transformation. Such a node can be depicted as:

Node C in this graph will be visited twice during the graph traversal; once from A and once from B. As C is a constrained node, it will only be rendered when all its constraints are satisfied. In other words, once C has been visited from both A (with A’s transformation) and B (with B’s transformation), the constraint will be valid and C can be rendered with some combination of the two transformations. Any children of C will inherit this compound transformation.

To return to the camera example, it might be that A supplies the position of the camera and B supplies the position of the target object. Once both have been established, the final camera transformation can be calculated.

The constraint that is attached to the node has references back to the nodes A and B so that it can recognize which nodes are supplying which transformations. The BSG provides some shortcuts for creating this arrangement automatically. To install a constraint, for example:

ConstraintLerpPositionHandle constraint(New);

// Constrains C to lie half way between A and B.

constraint->Install(constraint, C, A, B, 0.5f);

This will connect A and B to C and will connect the constraint to A and B, and then installs the constraint in C.

Constraints are triggered each time that they are satisfied, so if a portion of the graph with a constrained node is shared, it is possible that the constraint will fire several times analogous to the way that sub-graphs are handled without constraints. However, be careful! This:

Will visit C via A via X, then again via Y and A (overwriting the previous transform). When C is visited from B, the constraint will be satisfied and the final transformation derived from Root->Y->A->C and Root->B->C, so X will not have any influence on the result. Similarly:

will satisfy the constraint once Root->X->A->C and Root->B->C. But in this case Y->A->C only satisfies half the constraint which will leave the constraint in an unexpected state. It is best to avoid situations like this as they may lead to incorrect behavior.

## Callbacks

Sometimes it is useful to know what is going on during the rendering of the scene graph. The BSG provides a number of call-back hooks to allow an application to interact with the rendering process.

### Materials

Each time a material is used, a “draw callback” can be invoked. Draw callbacks are very useful for reducing the number of materials needed in a scene. Create draw callbacks by deriving from the base class DrawCallback e.g.

class ColorCallback : public DrawCallback

{

public:

ColorCallback(const Vec4 &color, MaterialHandle &mat) :

m\_color(color),

m\_uniform(mat->GetUniform<Vec4>("u\_color"))

{}

virtual bool OnDraw()

{

m\_uniform.Set(m\_color);

return true;

}

private:

Vec4 m\_color;

AnimTarget<Vec4> &m\_uniform;

};

This callback can be used to set the material color each time it is invoked. So, rather than have a large number of different materials each with a unique color, we can have a single material and modify the color when needed. BSG detects when the value of a uniform has changed, so will not needlessly update uniform values. Notice that the callback caches the animation target for the uniform when it is created. This is for efficiency to avoid looking up the target every time OnDraw() is called.

Draw callbacks are installed like this e.g.:

geom->AppendSurface(surf, mat, new ColorCallback(color, mat));

The callback is owned by the scene graph and will be deleted automatically.

### Scene Nodes

Callbacks can be added to scene nodes. Scene node callbacks define the following methods:

virtual bool OnRender(SemanticData &semData) const = 0;

virtual void OnModelMatrix(const Mat4 &xform) const = 0;

virtual void OnModelViewMatrix(const Mat4 &xform) const = 0

* The OnRender() method is invoked immediately prior to rendering the geometry in the scene node. Applications can modify the semantic data to override the transformations constructed by the scene-graph traversal. This can be useful, for example, to implement a billboard-like behavior where we might use the graph to position objects, but then override their transformations to make them orthogonal to the screen
* The OnModelMatrix() method is called when the model matrix has been calculated for the attached node. The application cannot change the transformation, but can record it or use it to calculate its own data.
* The OnModelViewMatrix() behaves as the OnModelMatrix() method, but reports the model-view matrix (combining the camera transformation with the model matrix).

Applications can write their own callback classes by deriving from the scene-node callback base classes. The BSG includes a SceneNodePositionCallback which can be attached to nodes to report their position.

Note that if a node is shared, it will generate multiple callbacks. A scene node may have zero or more callback objects attached. Callback objects are owned by the scene graph and are deleted automatically when the owning node is deleted.

### Cameras

A single callback of type CameraCallback may be attached to a camera. Camera callbacks report the view (and its inverse) transformation via the OnViewTransform() method:

virtual void OnViewTransform(const Mat4 &view, const Mat4 &iview) = 0;

Applications should define their own callback objects derived from the camera-callback class. It is useful to have access to the view information in order to set up light positions. New callbacks should be installed as heap allocated objects e.g.

m\_camera->SetCallback(new MyCameraViewCallback);

As with the other kinds of callback, the camera callbacks are deleted when the owning camera is deleted.

## Level of Detail

Every triangle that is drawn implies a certain overhead. When rendering complex objects which occupy only a small region of the screen, most of the triangles will not contribute to the final scene and so represent wasted effort.

To help alleviate this problem, the BSG provides a level-of-detail (LOD) system which automatically chooses between different versions of an object according to the screen area that it will occupy. As with view-frustum culling, this uses the bounding volume of the geometry. If the bounding volume is inaccurate, the results of the LOD calculations will also be wrong.

To create a model with level of detail, follow this procedure:

* Append the geometries for each LOD level to a node
* Use the LODGroup class to set the LOD information in the geometries

For example:

node->AppendGeometry(geom1);

node->AppendGeometry(geom2);

node->AppendGeometry(geom3);

LODGroup lodGroup;

lodGroup.Begin();

lodGroup.Add(geom1, 200.0f);

lodGroup.Add(geom2, 50.0f);

lodGroup.Add(geom3);

lodGroup.End();

This adds the three models to the node, and then describes how “geom1” should be used if its on-screen size is above 200 pixels. If it is smaller than 200 pixels, but bigger than 50 pixels, then “geom2” will be used. If it is smaller than 50 pixels, “geom3” will be used instead. The size is an approximate measure of the on-screen width or height of the object based on its bounding volume.

If the 3D scene is relatively simple, or if most objects are rendered at a reasonable size, it is probably not worth using the LOD system. Calculating the projected bounding box and selecting the appropriate model implies extra costly calculations. It is sensible to profile an application with and without the LOD system to see if it is an effective optimization.

The “hello\_lod” example is a simple application that demonstrates how LOD can be used:



# Multi-threading in BSG

BSG provides support for executing worker threads via two classes: the Tasker and the Task. An application uses the Tasker to launch threads of work and to monitor callbacks from them. Applications are not limited to a single Tasker object. See the “hello\_task” example for a simple use-case.

Declare the tasker in the application class or somewhere persistent:

Tasker m\_tasks;

To launch a task, requires code like this:

m\_tasks.Submit(new MyTask());

The task will be created and launched. When the slave thread completes, it automatically issues a callback which is placed in the tasker’s callback queue. Applications should periodically poll the callbacks via:

m\_tasks.DoCallbacks();

DoCallbacks() has an optional argument for the maximum number of callbacks to process. By default, all outstanding callbacks will be invoked.

The Task base-class has two overridable methods:

* OnThread()—executed in the slave thread. This is the work function for the task. Do not attempt to issue graphics calls (including, but not restricted to, calling methods such as RenderSceneGraph or texture object methods) in this thread. The main thread is associated with the GL context and calls from other threads will not work as expected.
* OnCallback() – executed in the main thread (typically after the slave has finished) during DoCallbacks().

Because applications have to explicitly create a Tasker there is no overhead for applications that do not wish to use multiple threads.

## Delay Loading Textures

Applications can use threads to “background” work that might take a long time. For example, loading a texture from a file could be a lengthy process, so it makes sense to submit a simple stand-in texture initially, launch a thread to load the real texture and get on with the main program. When the real texture has been loaded, it can be submitted and the application will pick it up on the next frame. The texture data must be submitted in the OnCallback() method and not in the OnThread() method because it requires a GL context.

The BSG TextureHandle class provides some helpers to simplify this work:

texture->TexImage2D(Image("Yellow.png", Image::eRGB888));

texture->TexImageTask(tasks, TexImage2D("Earth.png", Image::eRGB888));

This code will load the default “Yellow.png” texture, in the normal way, into the texture. The default will be used until the delay-loaded “Earth.png” is available. The TexImage2D helper is a Task which will perform the texture load on the slave thread, and will update the texture during its callback. The application needs to call tasks.DoCallbacks() to make sure that the update takes place.

## Asynchronous Callbacks

Slave threads do not have to wait until they complete before issuing their callbacks. Within an OnThread() method, you can call Callback() to inform the main thread that something has happened. As is usual with threads, care must be taken to ensure that any shared data is handled in a thread-safe manner (by using a Mutex or atomic operations, for example).

One possible application for this is to issue ticks back to the main application so that it can update a progress bar, for example. See the “hello\_task” example.

Handling the synchronization between threads can be complicated and error prone. For a mechanism more like regular function call see “Synchronous Callbacks” below.

## Mutex

BSG provides a simple Mutex class which has Lock() and Unlock() methods. It is used in the implementation of a thread-safe queue Queue<T> which is part of the task implementation, but can be used by multi-threaded applications to ensure thread safety. The performance of the Mutex object will be platform dependent.

To simplify the use of Mutex, BSG also implements a MutexLock object which locks on construction and unlocks on destruction.

## Event

The Event class has methods Wait() and Signal(). Events are created in an unsignalled state. Threads that wait on an event will wait indefinitely until the event is signaled on another thread. The Event class is used in the implementation of CallbackTask which is described in the next section.

## Synchronous Callbacks

The CallbackTask class, derived from Task, implements a common idiom for multi-tasking applications. Worker threads are forbidden from calling GL either directly or indirectly via BSG classes. Moreover, for efficiency, the BSG handle classes are not thread safe, so should only be manipulated in the main thread. This creates a problem for some kinds of workers. For example, the ObjLoaderTask worker can safely load data from a disk file, but cannot submit this data to GL. It could do all the work of submitting the data in the final callback, but for large objects this may introduce a significant delay resulting in the main thread pausing.

To solve this problem, the CallbackTask supports the notion of a *synchronous callback*. These look and behave almost the same as a regular method call e.g:

// In the worker thread OnTask method

Call(&MyCallbackTask::MyMethod, arg1, arg2, ...);

This will invoke MyMethod() with the specified arguments on the main thread and will wait until the method has completed before continuing on the worker thread. MyCallbackTask should be an application defined class derived from CallbackTask. CallbackTask implements the dispatch of the callbacks in its OnCallback method – you do not need to, nor should you override this method. The class should implement an OnTask() method as normal. Additionally, you can override the DefaultCallback() method which will be called when an asynchronous or terminal callback is issued.

The Call() mechanism currently supports up to five method arguments. The types of the arguments must match the corresponding arguments for the method. Arguments should be either values or pointers, as references are ambiguous.

Remember that callbacks are only serviced if the main thread calls Tasker::DoCallbacks.

# Overview of Effect Files

Effect files provide a way to describe how a surface should be rendered. Effect files contain an options section and a list of passes. Each surface will be rendered once for each pass. All the pass 0 rendering is completed before pass 1 is started.

## Options

The options section allows for the sorting order to be specified.

OPTIONS

{

SortOrder = NONE; // Or FRONT\_TO\_BACK or BACK\_TO\_FRONT or AUTO

}

Objects can be rendered either front-to-back, back-to-front, in an automatically defined order or in no particular order. It is recommended that opaque objects be rendered front-to-back because this maximizes the chances of occlusion by foreground objects. This in turn can lead to early rejection of obscured pixels saving work in shading these items.

However, transparent objects should be rendered back-to-front to obtain the correct blending behavior.

The Auto option will attempt to sort opaque objects front-to-back and transparent objects back-to-front.

The renderer will draw all front-to-back objects first, then all “none” objects and finally all the back-to-front objects.

The sort is based on the bounding volume of the object and is only approximate. Objects that overlap in their z-extents may be rendered “in the wrong order”. One solution to this problem is to split objects into multiple smaller surfaces.

## Passes

A pass in BSG is a combination of semantic declarations and vertex and fragment shader code. A pass might look like this:

PASS 0

{

SEMANTICS

{

u\_mvpMatrix = MATRIX4\_MODEL\_VIEW\_PROJECTION;

a\_position = VATTR\_POSITION;

a\_normal = VATTR\_NORMAL;

}

STATE

{

EnableBlend = false;

EnableDepthTest = true;

}

VERTEX\_SHADER

{

uniform mat4 u\_mvpMatrix;

attribute vec4 a\_position;

attribute vec3 a\_normal;

varying vec3 v\_normal;

void main()

{

v\_normal = a\_normal;

gl\_Position = u\_mvpMatrix \* a\_position;

}

}

FRAGMENT\_SHADER

{

precision mediump float;

uniform sampler2D u\_tex;

varying vec3 v\_normal;

void main()

{

gl\_FragColor = vec4(abs(v\_normal), 1.0);

}

}

}

In the semantics section, the names and types of uniforms and attributes should be declared. BSG supplies a set of semantic values such as MATRIX4\_MODEL\_VIEW\_PROJECTION and VATTR\_POSITION. The renderer sets the associated uniforms or attributes to the appropriate values at render time, so attributes declared as VATTR\_POSITION will be attached to the position attributes in surfaces, and the uniform declared as MATRIX4\_MODEL\_VIEW\_PROJECTION will be bound to the combined model-view-projection matrix. For variables with no special meanings, the BSG provides user semantics. It is the responsibility of the application to bind these to their appropriate values.

Sampler uniforms are handled in a different way to normal uniforms and are declared in their own section within the pass.

SAMPLER\_2D u\_tex

{

Unit = 0;

Wrap = CLAMP, CLAMP;

Filter = LINEAR\_MIPMAP\_NEAREST, LINEAR;

}

This arrangement allows the effect to specify the texture unit, wrap and filter modes as well as the name and kind of sampler involved.

In the state section, settings for the GL state appropriate for this shader should be declared. Having no declarations in this section applies the default GL state.

In the vertex and fragment shader sections, the code for the shaders should be included verbatim.

# Fonts and Text

The BSG provides two mechanisms for creating text. The first method is independent of the scene graph and provides a simple way to render 2D text. The second method provides 2D and 3D text whose position can be controlled by the scene graph itself. Both methods use the “freetype” font rendering library for glyph generation. Whilst BSG’s text rendering support is probably adequate for a GUI or similar application, it is not a fully featured font rendering system. Applications, like browsers, that handle a lot of text will probably need to implement their own solution.

See “hello\_text” for some examples of text rendering.



## Direct 2D Text

To render text using this method, an application must first create a font handle. This can then be used via the application DrawTextStringAbs() or DrawTextString() methods to render text directly onto the display.

void DrawTextStringAbs(const std::string &str, float xpos, float ypos,

FontHandle font, const Vec4 &color) const;

void DrawTextString (const std::string &str, float xpos, float ypos,

FontHandle font, const Vec4 &color) const;

DrawTextString() uses a normalized coordinate system (0..1) in x and y. The absolute variant uses exact pixel locations.

To create a font handle:

FontHandle font(New);

font->SetFontInPercent(fontName, fontSize, screenXres);

or:

font->SetFontInPixels(fontName, fontSizePixels);

One of the advantages of direct 2D text is that it is guaranteed to be pixel-aligned. This does mean that when its position is animated, it may look slightly jerky because it will snap to the nearest whole pixel. Direct 2D text is always orthogonal to the display and cannot be rotated. Because it is not placed in the scene graph it is difficult to align with 3D objects. It is the responsibility of the application to handle resizing and positioning of 2D text.

## Scene Graph Text

Direct 2D text is convenient for simple applications where the text position is unrelated to the 3D content. It is probably the best choice when creating HUD information or an information overlay. However when the text is, for example, a label on a 3D object or when the text itself is to be considered a 3D object, an application will need to use a different approach.

Scene graph text is also more flexible because it allows for custom effects to be attached to the text which facilitates texturing of text or other shader effects such as gradient fills. The example application “hello\_text” shows scene graph text with custom effects in action.

To use scene graph text an application must first create a “PrintFont” handle.

PrintFontHandle font(New);

font->Load(fontName, fontSize);

Using the font, we can set up a “Print” handle.

PrintHandle printGeom(New);

printGeom->SetFont(font);

printGeom->SetText("Hello 3D Text", 10.0f);

PrintHandles are a specialization of GeometryHandles, so they can be added to the scene-node geometry list in the same way as the geometry generated via e.g. shape factory classes.

By default, the text is white. This can be changed via:

printGeom->SetUniform("u\_textColor", color);

The example also shows how to create customized effects for the text.

One trick for maintaining 3D text orthogonal to the display is to place it below the camera. Any transformations that are applied to the camera will then be automatically inherited by the text. A translation/position can be used to place the text in the desired location on the display, and is one way to control the text size.

The way that text is positioned can be controlled using PrintOptions. These give control over where the “hotspot” of the text is positioned. The PrintOptions::eAnchor type enumerates the nine possible locations (eTOP\_LEFT, eTOP\_RIGHT, eBOTTOM\_LEFT, eBOTTOM\_RIGHT, eCENTER, eTOP\_CENTER, eBOTTOM\_CENTER, eCENTER\_LEFT or eCENTER\_RIGHT). There are two variations on calculating these hotspots. The first (and default) is best suited to 2D line-oriented text rendering; the second uses a bounding box of the text to calculate the hotspots.

printGeom->SetText("Hello 3D Text", 10.0f, eTOP\_CENTER);

When text is rendered BSG calculates a top, left, bottom and right for it. In the case of bounding-box mode, the text is treated as an array of character rectangles, and the box is the union of the bounds of all these rectangles.

When a bounding box is not used (the default), the top is always defined to be on the baseline of the first character in the string. The bottom is aligned to the baseline of the last line of characters. Left is zero and right is defined as in the bounding-box case.

## Modifying Text Behavior

Scene-graph text can be modified using a formatter callback. This is most useful for justifying or truncating text prior to rendering.

PrintHandle printGeom(New);

printGeom->SetFormatter(formatter);

printGeom->SetFont(font);

Formatters look like this:

class MyFormatter : public PrintFormatter

{

public:

MyFormatter(/\* params \*/);

virtual const std::vector<uint32\_t> &Format(const std::vector<uint32\_t> &str);

};

That is, the formatter is responsible for generating a new version of the input “string”. Characters are 32-bits wide to allow for international font support. The formatter can use the following base class methods to query the font metrics:

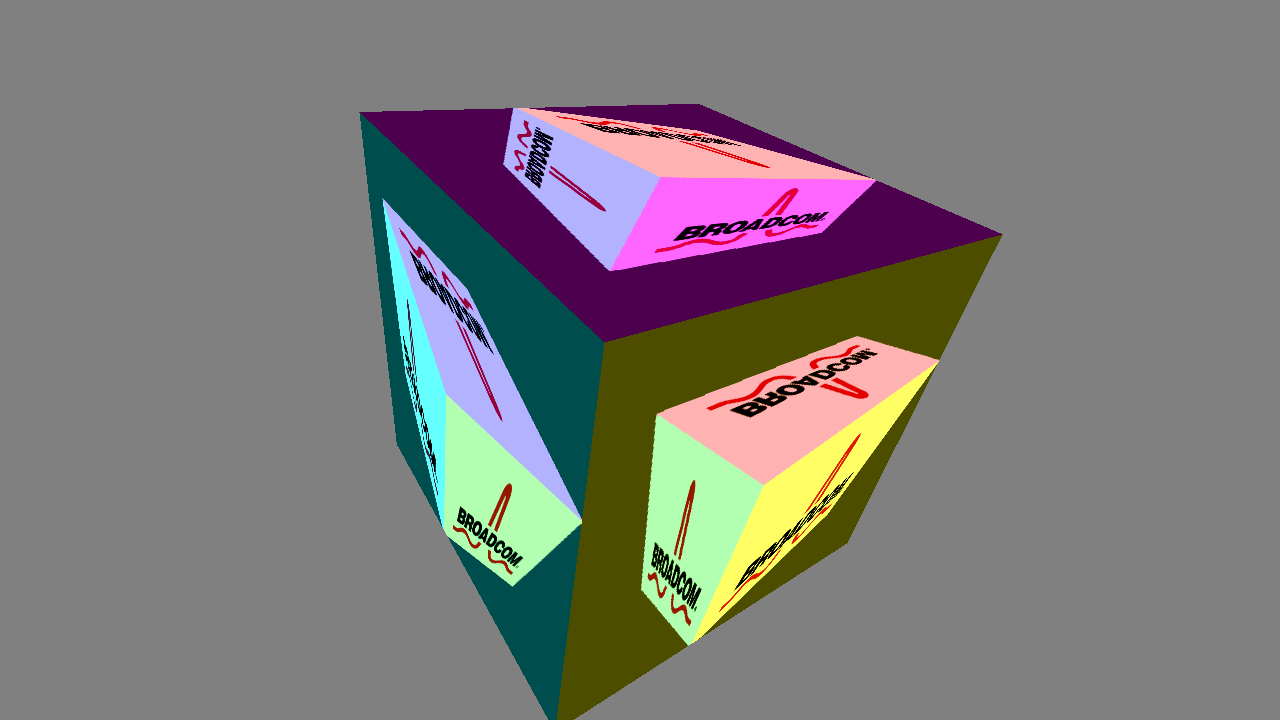
float GetAdvanceX(uint32\_t ch) const;

PrintCharInfo GetCharInfo(uint32\_t ch) const;

# Off-screen Rendering

The BSG supports rendering to framebuffer objects (FBOs). The most common use case for FBOs is to render to a texture. For example, an application modeling reflections might render a scene from the viewpoint of an object, say a coffee pot, into the faces of a cube map. The shader for the coffee pot could then use this cube map to look-up the reflected scene.

Using FBOs can cause a significant overhead. One reason for this is that the data in the color, depth and stencil buffers for both the FBO and default framebuffer have to be preserved when switching between render targets (but see the discard extension below).

See the “hello\_fbo” application for an example of using FBOs.

Like textures, FBOs in BSG are managed via handles so are created as:

FramebufferHandle my\_fbo(New);

For an FBO to be useful, we need to attach buffers to (at least one of) the depth, stencil and color attachment points. Depth and stencil attachments should be created as render-buffer objects. The color buffer attachment can be a render-buffer or a texture.

RenderbufferHandle my\_rbo(New);

Once a render-buffer object handle has been created, it can be sized:

my\_rbo->Storage(GL\_DEPTH\_COMPONENT16, WIDTH, HEIGHT);

To attach render buffers or textures to the framebuffer object use:

my\_fbo->Attach(Attachment(my\_texture, GL\_COLOR\_ATTACHMENT0),

Attachment(my\_rbo, GL\_DEPTH\_ATTACHMENT));

And to check that the FBO is properly configured use:

GLenum err = my\_fbo->Check(); // err == GL\_FRAMEBUFFER\_COMPLETE means OK

To use the framebuffer as a render target use the Application method SetRenderTarget():

SetRenderTarget(&my\_fbo);

And to return to the default buffer:

SetRenderTarget();

While an FBO is the render target, all rendering operations (including clears) will apply to the buffers in the FBO rather than to the default (on-screen) framebuffer. Do not destroy the FBO whilst it is the current render target.

An application could subvert the BSG framebuffer mechanism by using explicit calls to GL APIs. This is acceptable only if it doesn’t try to mix the two mechanisms. Mixing application managed FBOs and BSG FBOs is unlikely to work as expected.

## Framebuffer Discard

If it is known *a priori* that some of the contents of frame-buffers are no-longer needed, then they can be marked for discard using the GL\_EXT\_discard\_framebuffer extension. This extension is implemented in the Broadcom drivers but may not be present on all platforms.

This API can significantly improve performance when using FBO rendering.

The BSG supports framebuffer-discard through the FrameBuffer object. Once the object is initialized, you can mark buffers as discardable via:

my\_fbo->Discardable(GL\_DEPTH\_ATTACHMENT, GL\_STENCIL\_ATTACHMENT);

Once the buffers are no longer needed, issue the call to the method:

my\_fbo->Discard();

# Mathematical and Utility Classes

Most 3D applications will need to process data in the form of vectors (for positions and directions), matrices (for transformations) and sometimes quaternions (for rotations).

The BSG supplies implementations of these data-types which it uses in communicating with the application. Applications can also use these types for their own purposes.

## Vectors

The following types are available:

* Vec2, Vec3, Vec4 – vectors of floating point values
* IVec2, IVec3, IVec4 – vectors of integers

The template automatically generates most of the simple element-wise operations over these types. Specialized operations such as cross-product are also defined by the BSG.

## Matrices

Matrices work in a similar way to vectors. The framework defines matrices:

* Mat2, Mat3, Mat4

The latter two are used extensively in BSG for calculating transformations during graph traversal.

Operations between matrices and vectors treat the vectors as columns and which are therefore post-multiplied. This means that in:

the vector is first transformed by ***N*** then ***M***.

## Quaternions

Quaternions are a mathematical tool for representing rotations. They are a generalization of complex numbers. A quaternion can be described by four real numbers *(x, y, z, w)* and has correspondingly four components (a real component and three imaginary components):

BSG provides a small set of operations that can be applied quaternions:

// Constructors

Quaternion();

Quaternion(float degrees, float x, float y, float z);

Quaternion(float degrees, const Vec3 &axis);

Quaternion(const Vec4 &v);

void Normalize();

Quaternion SLerp(float t, const Quaternion &lerpTo) const;

void GetAxisAngle(float \*degrees, Vec3 \*axis) const;

void SetAxisAngle(float degrees, const Vec3 &axis);

void AsMatrix(Mat4 \*mx) const;

Quaternion operator\*(const Quaternion &rhs) const;

BSG also provides methods to access the fields of the quaternion.

## Transforms

As already described, scene nodes in the BSG scene graph have a transformation associated with them. Whilst it would be possible to represent this transformation directly as a matrix (Mat4), this is not convenient when modifying the transformation e.g. when animating.

Instead, the BSG represents each transformation in terms of three simpler operations:

* A translation **T** or position
* A rotation **R** or orientation
* A scale factor **S**

Transformations can be converted into matrix form as required, but the inverse operation is not supported (it is a complicated function which is ambiguous, so that converting a transform to a matrix and then vice versa will be unlikely to return the original transformation).

The matrix representing the compound transformation is. When post multiplied by a vector, this transformation scales, then rotates and then translates.

## Exceptions

The BSG defines its own exception type Exception which has a payload of a string. If the BSG encounters problems, it will throw exceptions of this type.

The Exception class has one method Message() for accessing the string. BSG also defines a utility macro BSG\_THROW. This can be used to throw a BSG Exception and can be supplied with an expression which is treated as an STL string e.g.

BSG\_THROW("Cannot find include file '" << path << "' at line " << m\_lineNumber);

## Time

The Time class is used by the animation subsystem, and can be used more generally by applications. Times are stored internally as a 64-bit integer number of milliseconds and can be treated as either absolute times (i.e. relative to an epoch), or as a duration. This difference is not enforced by the time class. It is a matter of interpretation in the application.

The Time class has a rich set of constructors to allow times to be specified in various units and various types.

//Relative times

Time(int32\_t val, eTimeUnit u = eMILLISECONDS);

Time(uint32\_t val, eTimeUnit u = eMILLISECONDS);

Time(int64\_t val, eTimeUnit u = eMILLISECONDS);

Time(float val, eTimeUnit u = eSECONDS);

Time(double val, eTimeUnit u = eSECONDS);

Time(uint32\_t secs, uint32\_t usecs);

// Times since epoch

Time(uint16\_t year, uint8\_t month, uint8\_t day,

uint8\_t hour, uint8\_t minute, uint8\_t second);

Time FromCalendarTime(time\_t calTime);

The BSG defines a set of useful operators on times:

// Arithmetic

Time operator+(const Time &rhs) const;

Time operator-(const Time &rhs) const;

Time operator\*(float scale) const;

// Comparisons

bool operator==(const Time &rhs) const*;*

bool operator!=(const Time &rhs) const*;*

bool operator<(const Time &rhs) const*;*

bool operator<=(const Time &rhs) const*;*

bool operator>(const Time &rhs) const*;*

bool operator>=(const Time &rhs) const*;*

and some conversion functions to map times to different time units:

int64\_t Milliseconds() const*;*

double Seconds() const*;*

double Minutes() const*;*

double Hours() const*;*

float FloatSeconds() const*;*

float FloatMinutes() const*;*

float FloatHours() const*;*

The float methods need to be used with some caution as a duration of over approximately 4 hours cannot be accurately represented in a floating-point variable.

The following methods convert an absolute time into the calendar equivalents:

time\_t CalendarTime() const;

uint16\_t CalendarYear() const; // e.g. 2012

uint8\_t CalendarMonth() const; // 1 = Jan

uint8\_t CalendarDay() const; // 1 to 31

uint8\_t CalendarHour() const; // 0 to 23

uint8\_t CalendarMinute() const; // 0 to 59

uint8\_t CalendarSecond() const; // 0 to 59

std::string CalendarTimeString(const std::string &format =

std::string("%d %B %Y - %X")) const;

The static method Time::Now() returns the current time relative to the start of the application run. It is almost always better to use the Application::NowTimestamp() method to get the current time, since this will be appropriately scaled if any rate-multiplier is in effect.

# Video

The BSG has a class for loading (from a data file) and playing-back video. To open a video file, simply construct a Video object via:

Video(const std::string &videoFileName);

The constructor will throw an exception if there is a problem. The behavior of the video decoder can be configured via:

void SetOutputParameters(uint32\_t width, uint32\_t height, eVideoFrameFormat format,   
 GLTexture::eVideoTextureMode mode);

void SetOutputFormat(eVideoFrameFormat format, GLTexture::eVideoTextureMode mode);

And individual frames retrieved via:

eFrameStatus GetFrame(eMode mode, VideoBuffer \*buffer);

The data will be decoded into the buffer. If mode is set to eBLOCK, the decoder will wait for a new (i.e. different to previously returned) frame. If it is eNO\_BLOCK, the current frame will be delivered.

The VideoBuffer is constructed using:

VideoBuffer(uint32\_t width, uint32\_t height, eVideoFrameFormat format,   
 NativePixmap \*nativePixmap)

And the native pixmap is created via:

NativePixmap(uint32\_t w, uint32\_t h, ePixmapFormat format);

*Note*: Video is not currently supported when using a Nexus surface compositor based platform due to individual frames being inaccessible over NSC. This restriction will be lifted in the future.

# GUI Classes

## Carousel

To help with the construction of GUI applications, BSG provides a “carousel” class. The carousel is a general GUI element. It consists of a rotating set of M items of which a window of N will be visible at any one time. Each item is associated with some drawable elements and a transformation. The carousel can be made to “rotate”. The effect is to move the elements in a cyclic fashion so that for a carousel consisting of items A, B, C, D, E with B, C, D being visible, a rotation in one direction will make A visible and move it to the position of B, B will move to C, C to D and D will move to E and become invisible. The carousel can be rotated in the opposite direction with the reverse effects. This behavior is useful for, but not restricted to, implementing a rotating carousel effect (hence its name) as in the test application example.

The carousel class smoothly interpolates between the positions over a specified time period. During the animation, the carousel will ignore any further requests to rotate. Applications can inquire the IsAnimating() state to determine if the carousel is “busy”.

The carousel uses the BSG scene graph machinery internally, so each item in the carousel can be an arbitrarily complicated object. They can even be animated. Moreover, the root node of the carousel can be embedded in a scene graph so that the whole carousel can be transformed.

The carousel will handle instances when the number of objects is less than the number that are displayed by replication.

To specialize the carousel, it is recommended that applications derive their own carousel class. The constructor can create the “shape” of the carousel by adding transformations into the carousel via e.g.

std::vector<Transform> trans;

for (uint32\_t i = 0; i < NUM\_TRANS; ++i)

{

trans.push\_back(Transform(i));

}

SetLayout(trans);

There should be a transformation for each physical position in the carousel layout. The SetLayout() method will accept any suitable container (it should support a bidirectional iterator). Note that the first and last positions are special. Objects moving into/out from these positions can optionally be faded out/in. This option is specified in the class constructor for the carousel.

The SetLayout() method takes an optional Boolean argument “loop”. If this is set to true, then SetLayout() creates a loop such that the first item will move to the last item position in one direction, and the last item will move to the first item position in the other direction.

The constructor or application can now inject the drawable elements that make up the carousel using the base class methods:

void AddGeometry(GeometryHandle geom); // Adds single piece of geometry

void AddGeometry(SceneNodeHandle geomNode); // Add a single graph

and controls which elements are displayed using:

virtual void Next(Time now, Time animationTime);

virtual void Prev(Time now, Time animationTime);

To use the carousel, an application should call UpdateAnimationList() during the UpdateFrame() callback.

To find the currently selected item, call the method:

uint32\_t CurrentIndex() const;

By default, the middle item (rounding down), is considered to be the selected item. This can be overriden by setting the selected position explicitly via:

void SetSelectedItem(int32\_t i);

Call this method during configuration of the carousel.

The “logo\_menu” example makes use of the carousel to build a simple menu UI.

# BSG Hints and Tips

This section contains some advice on how to get the most out of BSG.

## Assemblies

One key to successful software engineering is encapsulation. C++ goes a long way to help in this regard, but there are other things to consider when designing BSG-based classes. This section describes the notion of an “assembly”. An assembly isn’t a formal type or class, but it has certain properties that make it easier to integrate into a wider system. An assembly might:

* Export one or more scene graph root handles. Clients of the object can then insert this chunk of assembly graph into their own graph and render the item subject to their own transformation. Other handles, such as materials or effects can be similarly exported.
* Allocate once, populate many. Exported handles should persist for the lifetime of the object. This makes it safe for other parts of the software to link to the item without worrying that the underlying object might disappear. If the content is subject to change, nodes can be cleared and repopulated to change what they draw, but the handle itself should not be deleted or set to null.
* Anticipate animation and expose animatable targets where sensible.
* Assemblies should export animatable targets using the most generic type available. For example, you might have an animatable uniform representing a color. The assembly should expose it as an AnimTarget<Vec4> &, not as an animatable uniform. After all, it is possible that future developments might change the implementation to, say, an AnimatableGroup<Vec4>. By exposing the generic target type, the software outside of the assembly will need no changes.

## Performance Tips

BSG does a pretty good job of generating a small number of OpenGL ES calls per draw call. It does this through a combination of state-tracking and intelligent sorting. The following suggestions will help to make you applications as fast as possible:

* Remember that if nothing has changed in the scene and no animations are active, you can tell BSG to idle by setting the idleMs parameter of UpdateFrame(). There is no point re-rendering everything if nothing has changed.
* Broadcom 3D graphics cores can eliminate work by noticing when a pixel is going to be obscured by something that has already been rendered. To make the most of this, it is recommended to classify your effects appropriately using the sort-order. Opaque objects should be sent to the front-to-back list. Transparent objects should be flagged as back-to-front in order that blending works as expected. The AUTO sort mode can help here.
* View frustum culling can eliminate objects that are not visible to the camera. However, it is not free and if the visibility of objects is easy for the application to determine, it should do the work rather than rely on the generic BSG mechanism. Note also that view frustum culling relies on the bounds of objects being accurately set.
* Complex models can benefit from using the LOD system. However, note that if a suitable object complexity is easy for the application to determine, it should do the work itself.
* If textures are going to be projected on-screen at various sizes, use mip-mapping. Either load your own mip-maps, or get the BSG to generate them automatically. Mip-mapping is very useful in reducing texture bandwidth requirements. However, full tri-linear mip-mapping is more costly than simple bilinear (approximately 2x bandwidth cost).
* Every OpenGL call implies a certain overhead, so the less calls per draw, the better. BSG tries to minimize the number of state changes during scene traversal, but applications can help by giving their objects an explicit “sort priority” to gather objects with the same state together.

# Extending BSG

BSG is a framework and as such attempts to implement the most common functionalities required by applications. Inevitably this coverage will be incomplete. The framework is not a black box and it may be necessary to implement new features similar to but different from those provided in BSG.

There are several ways in which an application can extend or modify the functionality of BSG. Previous sections have already demonstrated how to install callbacks customize text rendering. The following sections show some other ways in which an application can extend BSG’s behavior. Each section is labeled “basic”, “intermediate” or “advanced”.

## Custom Geometry (basic)

The shape factory classes provide a set of geometry creation methods for various common geometric shapes. Applications can derive their own factory classes using the StdFactory base class as a starting point, or they can generate their own custom geometry.

To create a new standard factory, derive a class from Stdfactory:

class MyFactory : public StdFactory

{

public:

MyFactory(/\* parameters\*/);

// Only required if this is an indexed primitive

virtual void PushIndices(std::vector<short> &indices) const;

};

The constructor should create the geometry. The StdFactory has a number of AddVertex() methods to help with creating the shape.

MyFactory::MyFactory(...) :

StdFactory(GL\_TRIANGLE\_STRIP, NUM\_VERTICES\_HINT, CullMode(GL\_CW, GL\_BACK))

{

...

AddVertex(x0, 0.0f, y0, 0.0f, 1.0f, 0.0f, u0, v0, axis);

...

}

If this is an indexed primitive then the PushIndices() method should be implemented e.g.

void MyFactory::PushIndices(std::vector<short> &indices) const

{

indices.push\_back(0);

indices.push\_back(7);

indices.push\_back(2);

...

}

The StdFactory framework will handle the implementation of MakeSurface() and MakeGeometry() automatically.

## Adding Animation Types (intermediate)

Adding new kinds of interpolator or evaluator to create new kinds of animation is a straightforward process. To create a new interpolator:

class MyInterpolator : public BaseInterpolator

{

public:

virtual float GetInternalParamValue(const Time &time) const

{

return ...;

}

};

The method GetInternalParamValue() should use its “time” argument to generate some interpolated value according to its own interpolation scheme. Interpolators must be 0.0 at the start time of the animation and 1.0 at the end time. In between, or for extrapolated times outside the start and end range, values smaller than 0.0 or greater than 1.0 are acceptable.

For example a linear interpolation might look like this:

virtual float GetInternalParamValue(const Time &time) const

{

return (float)((time – m\_startTime).Seconds() / m\_duration.Seconds());

}

You might also call the CalcParam() method within your GetInternalParamValue() method to generate an initial linear interpolation value as it uses 64-bit time values for accuracy over long time periods and then converts to float.

By applying various mathematical functions to the standard linear interpolated value you can effect various types of easing curve. Many of these are already built into BSG, but you can create your own custom easing curves this way. Here’s how a Hermite easing curve is implemented in BSG:

virtual float GetInternalParamValue(const Time &time) const

{

float t = CalcParam(time); // Gets a standard accurate lerp value

float t2 = t \* t;

t = 3.0f \* t2 - 2.0f \* t2 \* t;

t2 = t \* t;

t = 3.0f \* t2 - 2.0f \* t2 \* t;

return t;

}

So, by adding a new interpolator, you can customize how an animation is interpolated between its start and end values. An Evaluator is used to take the interpolator’s generated parameter value (usually in the range 0 to 1) and generate a concrete type somewhere between the start and end values.

Adding a new evaluator follows a similar pattern to the interpolator. All evaluators derive from the base class Evaluator<T> where T is the type of the result. To create a new evaluator, for example:

class MyEvaluator : public Evaluator<Vec3>

{

public:

virtual Vec3 Evaluate(float p) const

{

return (m\_start \* (1.0f - p)) + (m\_end \* p);

}

};

The evaluate function should calculate a standard linear interpolation between the start and end values using the parameter p. Any non-linear interpolations are already encoded into the p value by the interpolator, so the evaluator does not need to do anything else.

By convention, the initialization method for evaluators is called Init(). This typically just records the start and end values in the appropriate type for the evaluator.

Finally a new animation binding type can be defined:

typedef AnimBinding<MyEvaluator, MyInterpolator> AnimBindingMine;

Evaluators and interpolators must support default construction.

## Derived Handles (advanced)

Applications can extend the existing handle types via an inheritance relationship. Because the handle types are a kind of smart pointer, the mechanism is slightly more involved than a simple derived class.

Suppose that an application wishes to create a specialized SceneNodeHandle called MySceneNodeHandle. Sitting behind these handles are the concrete types SceneNode and MySceneNode, and the latter will inherit from the former:

class MySceneNode : public SceneNode

{

public:

void NewMethod();  
 };

We need to define a corresponding handle type for our new class:

typedef Handle<MySceneNodeTraits> MySceneNodeHandle;

The key to making this work with BSG is to define the traits class to encode the relationship between MySceneNodeHandle and SceneNodeHandle.

struct MySceneNodeTraits

{

typedef SceneNode Base; // Reflect MySceneNode : public SceneNode

typedef MySceneNode Derived; // into BSG

typedef SceneNodeTraits BaseTraits; // The base handle traits

};

The BSG now has the information that it needs in order to be able to convert safely between a MySceneNodeHandle and a SceneNodeHandle. Handles should be allocated in the normal way:

MySceneNodeHandle mine(New);

Note that just as with ordinary pointers, once a handle has been converted to its base handle type, you will no longer have access to any derived handle methods (unless they are virtual in the base class).

SceneNodeHandle base = mine; // Just as with real pointers, this is OK

mine->NewMethod(); // OK

base->NewMethod(); // Invalid

## Adding Constraints (advanced)

Constraints are managed via handles. To extend the repertoire of constraints, the first step is to create a derived handle class as described in the previous section. For example, to create a new constraint handle:

struct MyConstraintTraits

{

typedef Constraint Base;

typedef MyConstraint Derived;

typedef ConstraintTraits BaseTraits;

};

typedef Handle<MyConstraintTraits> MyConstraintHandle;

The base class Constraint is abstract and is defined as:

class Constraint : public RefCount

{

public:

virtual ~Constraint() {}

// Returns true if all the constraints have been satisfied for a particular render

virtual bool Ready() const = 0;

// When the constrained node is visted, the constraints are passed the node handle

// and the matrix transform of its parent.

virtual bool Visit(const SceneNode \*who, const Mat4 &mat) const = 0;

// Combine this constraint's transformation

virtual void Apply(Mat4 &trans) const = 0;

protected:

// Hide constructor

Constraint() {}

private:

};

The derived class must implement the pure virtual functions Ready(), Visit() and Apply(). The BSG uses Ready() to determine when the constraint has been satisfied. Visit() is called by the framework when the constrained node is visited during graph traversal. Apply() is responsible for generating the transformation matrix for the constraint.

All built-in constraints also provide an Install() method which helps to attach a constraint to its node and to the constraining nodes. This method is not virtual as the interface will be different for each concrete constraint.

To help with writing the constraint machinery, the BSG provides a template class called Memo<T>. This class acts as a temporary store with the following operations:

//! Is there a valid value set

bool IsSet() const;

//! Get the value and clear it

const T &GetInvalidate() const;

//! Set the value

void Set(const T &value) const;

//! Clear the value

void Clear();

The Memo class should be used to cache the data from each visit to the constraint. When all the memo values in a constraint have been filled, then the constraint is satisfied.

The following shows how a constraint might be implemented. Given the following member declarations:

SceneNode \*m\_node1; // Constraint depends on this node

SceneNode \*m\_node2; // and this one

AnimatableFloat m\_alpha; // This is a control to allow animation

Memo<Vec3> m\_memo1; // Remembers position of node1

Memo<Vec3> m\_memo2; // Remembers position of node2

The implementation would be:

//! Have both the parents supplied values yet?

bool MyConstraint::Ready() const

{

return m\_memo1.IsSet() && m\_memo2.IsSet();

}

//! Remember the value from a parent node.

//! Return true if all values have been acquired.

bool MyConstraint::Visit(const SceneNode \*who, const Mat4 &xform) const

{

if (who == m\_from)

m\_memoFrom.Set(GetTranslation(xform));

if (who == m\_to)

m\_memoTo.Set(GetTranslation(xform));

return Ready();

}

//! Set the transform from the constraint

void MyConstraint::Apply(Mat4 &trans) const

{

const Vec3 &from = m\_memoFrom.GetInvalidate();

const Vec3 &to = m\_memoTo.GetInvalidate();

Vec3 interp = from \* (1.0f - m\_alpha) + to \* m\_alpha;

SetTranslation(trans, interp);

}

Finally, the Install() method would look like this:

void MyConstraint::Install(

MyConstraintHandle thisHandle, // Our own handle

SceneNodeHandle &owner, // The node which will be constrained

SceneNodeHandle &node1, // One constraining node

SceneNodeHandle &node2, // Another constraining node

float alpha) // Initial interpolation parameter

{

m\_from = from.Ptr();

m\_to = to.Ptr();

m\_alpha.Set(alpha);

m\_memoFrom.Clear();

m\_memoTo.Clear();

m\_from->AppendChild(owner);

m\_to->AppendChild(owner);

owner->AppendConstraint(thisHandle);

}