# Mobile 3D Graphics API

## Technical Specification

Version 1.1

June 22, 2005

Java Community Process (JCP)
JSR-184 Expert Group

# Contents

# Overview

## Preface

This document contains the specification of the Mobile 3D Graphics API (abbreviated "M3G") for the Java 2 Platform, Micro Edition ("J2ME"). The specification was defined within the Java Community Process ("JCP") under Java Specification Request 184 ("JSR-184"). The specification is subject to the terms of the JCP agreements (i.e. JSPA and/or IEPA).

The Mobile 3D Graphics API is an *optional package*. An optional package can be adopted to existing J2ME *profiles*. A profile of J2ME defines device-type-specific sets of APIs for a particular vertical market or industry. The main target platform of this optional API is J2ME/CLDC, used with profiles such as MIDP 1.0 or MIDP 2.0. However, the API can also be implemented on top of J2ME/CDC, or any Java platform in general.

Technical details of the API can be found in the package overview and the individual class descriptions; see especially the Graphics3D class. To see how the API is used in practice, refer to the example MIDlets at the end of the Package overview.

This specification uses definitions based upon those specified in RFC 2119 (available on the IETF web site).

| Term | Definition |
|---|---|
| MUST | The associated definition is an absolute requirement of this specification. |
| MUST NOT | The definition is an absolute prohibition of this specification. |
| SHOULD | Indicates a recommended practice. There may exist valid reasons in particular circumstances to ignore this recommendation, but the full implications must be understood and carefully weighed before choosing a different course. |
| SHOULD NOT | Indicates a non-recommended practice. There may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label. |
| MAY | Indicates that an item is truly optional. |

## Background

The objective of the Mobile 3D Graphics API Expert Group was to provide an efficient 3D Graphics API suitable for the J2ME platform, in particular CLDC/MIDP. The API is targeted at CLDC class of devices that typically have very little processing power and memory, and no hardware support for 3D graphics or floating point math. The API has been defined such that implementations on that kind of hardware are feasible. However, the API also scales up to higher-end devices featuring a color display, a DSP, a floating point unit, or even specialized 3D graphics hardware.

The M3G specification is based on the requirements, summarized below, that were agreed on by the Expert Group. The rationale for each requirement is presented in the paragraphs following the summary.

- The API **must** support retained mode access (that is, a scene graph).
- The API **must** support immediate mode access, with features similar to OpenGL.

- The API **must** support mixing and matching of immediate and retained mode access.
- The API **must not** include optional parts (that is, all methods must be implemented).
- The API **must** have importers for meshes, textures, entire scene graphs, etc.
- The API **must** be efficiently implementable on top of OpenGL ES.
- The API **must** use the native float data type of Java, not introduce a custom type.
- The API **must** be efficiently implementable without floating point hardware.
- The API **should** be implementable within 150 kB on a real mobile terminal.
- The API **must** be structured so that garbage collection is minimized.
- The API **must** be interoperable with related Java APIs, especially with MIDP.

Several applications were identified for the Mobile 3D Graphics API, including games, map visualization, user interfaces, animated messages, product visualization, and screen savers. Each of these have different needs: some require simple content creation, some require high polygon throughput, yet others require high quality still images with special effects.

It is clear that such a wide spectrum of different needs cannot be fulfilled by a scene graph API alone, nor an immediate mode API alone. It is also clear that having two separate APIs would lead to developer confusion and sub-optimal usage of precious memory space. Rather, there must be only one indivisible API, with only one RI and TCK, that covers both types of access in a unified way. A developer should be able to use either one, or both at the same time, depending on the task at hand.

The immediate mode (or low-level) part of the API should be a subset of OpenGL with no added functionality. That is, the low-level features should be compatible with OpenGL ES, which is being standardized by Khronos. For the Reference Implementation, the scene graph (or high-level) part must be built on top of the low-level interface, and shall never bypass it when rendering. This ensures that the scene graph does not include rendering features that cannot be implemented directly with the low-level interface. The low-level implementation may then be changed freely, or even accelerated with dedicated hardware.

In many cases, there is little else to an application than displaying a scene and playing back some animation created in a 3D modeling tool. This should not require much Java programming. Even in more demanding cases, it greatly speeds up development if it is easy to import objects and animations into a midlet. Therefore, the API must provide importer functions for different data types, such as textures, meshes, animations, and scene hierarchies. The data must be encoded in a binary format for compact storage and transmission.

Most mobile terminals do not have hardware support for floating point processing. This should be reflected in the API, so that it can be efficiently implemented using integer arithmetic. However, since programming with fixed point mathematics is difficult and error prone, the API should use floating point values wherever feasible, and plain integers otherwise. Fixed point values must not be used. Also, instead of introducing a custom Float type or packing floating point values into integers, Java's built-in float data type must be used. As a consequence, this API can not be implemented on top of CLDC 1.0.

Like all APIs targeting MIDP, we need to strive for as compact an implementation as possible. It should be possible to implement the API in less than 150 kB of ROM space, including the native graphics engine, Java class files (ROMized), and content importing facilities. To minimize garbage collection, the API should be structured so that using it does not require repetitive creation of objects.

The API must be tightly integrated with MIDP's LCDUI, such that 2D and 3D graphics can be efficiently rendered on the same Canvas or Image, for example. The decision of whether to use double buffering or not should be delegated to the MIDP implementation. The API should also be able to bind with other GUI APIs, such as AWT.

## Related Literature

- *The Java Language Specification*, James Gosling, Bill Joy, and Guy L. Steele, 1996.

- *Quaternion Algebra and Calculus*, David Eberly, 1999.
- *Key Frame Interpolation via Splines and Quaternions*, David Eberly, 1999.
- Connected, Limited Device Configuration (JSR-30), Sun Microsystems, Inc.
- Connected, Limited Device Configuration 1.1 (JSR-139), Sun Microsystems, Inc.
- Connected Device Configuration (JSR-36), Sun Microsystems, Inc.
- Mobile Information Device Profile (JSR-37), Sun Microsystems, Inc.
- Mobile Information Device Profile 2.0 (JSR-118), Sun Microsystems, Inc.
- OpenGL 1.3 Specification, Silicon Graphics, Inc.
- PNG file format, World Wide Web Consortium (W3C).

## Revision History

| Date | Version | Description |
|---|---|---|
| 22 Jun 2005 | 1.1 | Maintenance Release |
| 19 Nov 2003 | 1.0 | Final Release |

## Changes from version 1.0 to 1.1

New features:

- The Loader now supports all PNG color types and bit depths.
- The Node alpha factor now affects Sprite3D as well.
- Several `get` methods added to allow all properties to be queried.
- `OVERWRITE` hint flag added to `Graphics3D.bindTarget`.

Removed or relaxed exceptions:

- `Object3D.removeAnimationTrack` no longer throws NullPointerException.
- `Graphics3D.releaseTarget` no longer throws IllegalStateException.
- Removed several deferred exception situations in VertexBuffer.
- Largest possible target surface and viewport need no longer be square.
- `Group.addChild` no longer throws an exception if the Node is already a child of the Group.

New or tightened exceptions:

- Target surfaces larger than maximum viewport are no longer allowed in Graphics3D.

Resolved interoperability issues:

- Default projection matrix is now required to be identity, with projection type `GENERIC`.
- The Loader must now treat all file names as case sensitive.
- Mutable MIDP Images are treated as RGB, immutable Images as RGBA.
- Emphasized that flipping the sign of a quaternion when interpolating is not allowed.
- Downscaling of sprite and background images is now well specified.
- Clarified the role of the crop rectangle with scaled sprites.

## Acknowledgements

The Mobile 3D Graphics API (JSR-184) Expert Group member companies are listed in the table below, as well as the names of those company representatives who contributed to the version 1.0 specification.

| Contributors | Member Companies |
|---|---|
| Tomi Aarnio, Nokia **(Editor)** | Aplix |
| Dirk Ambras, Siemens | ARM |
| Paul Beardow, Superscape | Bandai Networks |
| Mark Callow, HI Corporation | Cingular Wireless |
| Frederic Condolo, In-Fusio | Cellon France |
| Sean Ellis, Superscape **(Associate Editor)** | France Telecom |
| Harri Holopainen, Hybrid Graphics | Fuetrek |
| Jyri Huopaniemi, Nokia **(Spec Lead)** | HI Corporation |
| James Irwin, Vodafone | Hybrid Graphics |
| Kari Kangas, Nokia | In-Fusio |
| Matti Kantola, Nokia | Insignia Solutions |
| Hidekazu Koizumi, Aplix | Intel |
| Ville Miettinen, Hybrid Graphics | Intergrafx |
| Hannu Napari, Hybrid Graphics | MathEngine |
| Mark Patel, Motorola | Motorola |
| Kari Pulli, Nokia | Nokia **(Spec Lead)** |
| Kimmo Roimela, Nokia **(Associate Editor)** | Research In Motion |
| Michael Steliaros, Superscape | Siemens |
| Jacob Ström, Sony Ericsson | Sony Ericsson |
| Mika Tammenkoski, Sumea | Sun Microsystems |
| Mark Tarlton, Motorola | Superscape |
| Doug Twilleager, Sun Microsystems | Symbian |
| Brian Young, Research In Motion | Texas Instruments |
| Lincoln Wallen, MathEngine | 3d4w |
| Simon Wood, Superscape | Vodafone |
| | Zucotto Wireless |

The following companies and individuals contributed to the version 1.1 specification:

| Contributors | Member Companies |
|---|---|
| Tomi Aarnio, Nokia **(Spec Lead, Editor)** | Aplix |
| Paul Beardow, Superscape | ATI |
| Mark Callow, HI Corporation | HI Corporation |
| Sean Ellis, Superscape | Hybrid Graphics |
| Chris Grimm, ATI | Motorola |
| Ville Miettinen, Hybrid Graphics | Nokia **(Spec Lead)** |
| Kari Pulli, Nokia | Sony Ericsson |
| Kimmo Roimela, Nokia **(Associate Editor)** | Superscape |
| Keh-Li Sheng, Aplix | |
| Michael Steliaros, Superscape | |
| Jacob Ström, Sony Ericsson | |
| Mark Tarlton, Motorola | |
| Simon Wood, Superscape | |

# Package javax.microedition.m3g

Defines an API for rendering three-dimensional (3D) graphics at interactive frame rates, including a scene graph structure and a corresponding file format for efficient management and deployment of 3D content.

**See:**
**Description**

| Class Summary | |
|---|---|
| **AnimationController** | Controls the position, speed and weight of an animation sequence. |
| **AnimationTrack** | Associates a KeyframeSequence with an AnimationController and an animatable property. |
| **Appearance** | A set of component objects that define the rendering attributes of a Mesh or Sprite3D. |
| **Background** | Defines whether and how to clear the viewport. |
| **Camera** | A scene graph node that defines the position of the viewer in the scene and the projection from 3D to 2D. |
| **CompositingMode** | An Appearance component encapsulating per-pixel compositing attributes. |
| **Fog** | An Appearance component encapsulating attributes for fogging. |
| **Graphics3D** | A singleton 3D graphics context that can be bound to a rendering target. |
| **Group** | A scene graph node that stores an unordered set of nodes as its children. |
| **Image2D** | A two-dimensional image that can be used as a texture, background or sprite image. |
| **IndexBuffer** | An abstract class defining how to connect vertices to form a geometric object. |
| **KeyframeSequence** | Encapsulates animation data as a sequence of time-stamped, vector-valued keyframes. |
| **Light** | A scene graph node that represents different kinds of light sources. |
| **Loader** | Downloads and deserializes scene graph nodes and node components, as well as entire scene graphs. |
| **Material** | An Appearance component encapsulating material attributes for lighting computations. |
| **Mesh** | A scene graph node that represents a 3D object defined as a polygonal surface. |
| **MorphingMesh** | A scene graph node that represents a vertex morphing polygon mesh. |
| **Node** | An abstract base class for all scene graph nodes. |
| **Object3D** | An abstract base class for all objects that can be part of a 3D world. |
| **PolygonMode** | An Appearance component encapsulating polygon-level attributes. |
| **RayIntersection** | A RayIntersection object is filled in by the `pick` methods in Group. |
| **SkinnedMesh** | A scene graph node that represents a skeletally animated polygon mesh. |
| **Sprite3D** | A scene graph node that represents a 2-dimensional image with a 3D position. |

| | |
|---|---|
| **Texture2D** | An Appearance component encapsulating a two-dimensional texture image and a set of attributes specifying how the image is to be applied on submeshes. |
| **Transform** | A generic 4x4 floating point matrix, representing a transformation. |
| **Transformable** | An abstract base class for Node and Texture2D, defining common methods for manipulating node and texture transformations. |
| **TriangleStripArray** | TriangleStripArray defines an array of *triangle strips*. |
| **VertexArray** | An array of integer vectors representing vertex positions, normals, colors, or texture coordinates. |
| **VertexBuffer** | VertexBuffer holds references to VertexArrays that contain the positions, colors, normals, and texture coordinates for a set of vertices. |
| **World** | A special Group node that is a top-level container for scene graphs. |

## Package javax.microedition.m3g Description

Defines an API for rendering three-dimensional (3D) graphics at interactive frame rates, including a scene graph structure and a corresponding file format for efficient management and deployment of 3D content.

The function of this API is to provide Java application programmers with an efficient and flexible means to display animated 3D graphics in real time on embedded devices. To cater for the needs of different types of applications, both an easy-to-use scene graph structure and an immediate mode interface are provided. All animation and rendering features are available for scene graph objects and individually rendered objects alike. The developer therefore does not need to choose between the immediate mode and the scene graph, but rather can mix and match both within the same application.

Besides the API itself, a corresponding file format for efficient storage and transfer of all necessary data is also defined. This data includes meshes, textures, scene hierarchies, material properties, animation keyframes, and so on. Data is written into a file by content creation tools on a PC, and loaded into the API through the Loader class.

### Getting Started

The example applications at the end of this page provide a good means to get a quick overview of this API. Of the individual classes, `Graphics3D` is perhaps the most important, because all rendering is done there. The `World` class is crucial because it serves as the root of the scene graph structure. `Object3D` is the base class of all objects that can be rendered or loaded from a file, and also the place where animations are applied. We also recommend you to read the rest of this package description.

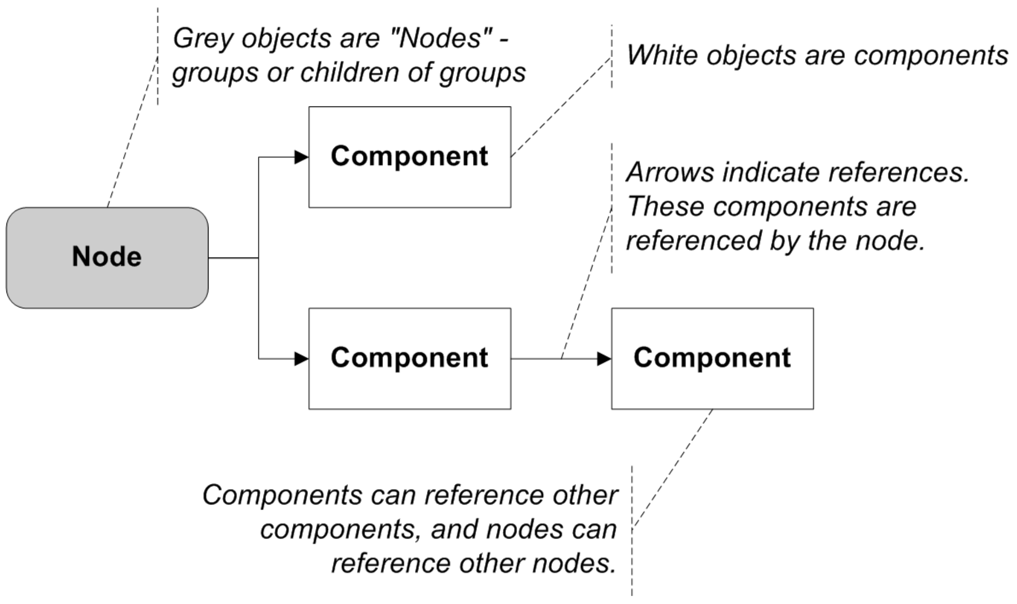
### Package Discovery

Because of its optional nature, this API may not always be available on every platform. Each profile and platform may have their own methods for J2ME package discovery as there is no universal method existing at this time. An additional method for package discovery of the Mobile 3D Graphics API is by using a system properties query. To discover this package, call `System.getProperty` with a key of `microedition.m3g.version`. If the API is present, the value returned is the version of the API (this version is "1.1", and the previous version was "1.0"). If the API is not present then the key is also not present and `null` will be returned.

## Documentation Conventions

The following general conventions are observed in the documentation of this API.

- **Coordinate systems.** All 2D coordinate systems follow the MIDP convention where the origin is in the upper left corner and integer coordinates are at pixel boundaries. By default, 3D coordinate systems are right-handed, and rotations obey the right-hand rule: looking along the positive axis of rotation, positive angles are clockwise. The camera coordinate system follows the OpenGL convention where the view direction coincides with the negative z-axis, the positive x-axis points right, and the positive y-axis points up. However, the application is free to set up left-handed 3D coordinate systems by use of transformation matrices.
- **Matrix notation.** Matrices are denoted as upper case bold letters, and vectors as lower case bold letters. For example, **M** denotes a matrix and **v** a vector. Matrices have 4x4 and vectors 4 elements, unless stated otherwise. Vectors are always column vectors, and are consequently on the right hand side when multiplied with a matrix: **v'** = **M v**.
- **Numeric intervals.** Closed intervals are denoted with square brackets and open intervals with round brackets. For example, [0, 10) denotes the values from zero to ten, including zero but not including ten. Depending on the context, a numeric interval may consist of real numbers or integers.
- **OpenGL references.** All references to OpenGL in this specification are to version 1.3. See Related Literature on the overview page.
- **Diagram notation.** The following common notation is used in diagrams that involve scene graph nodes and node components.

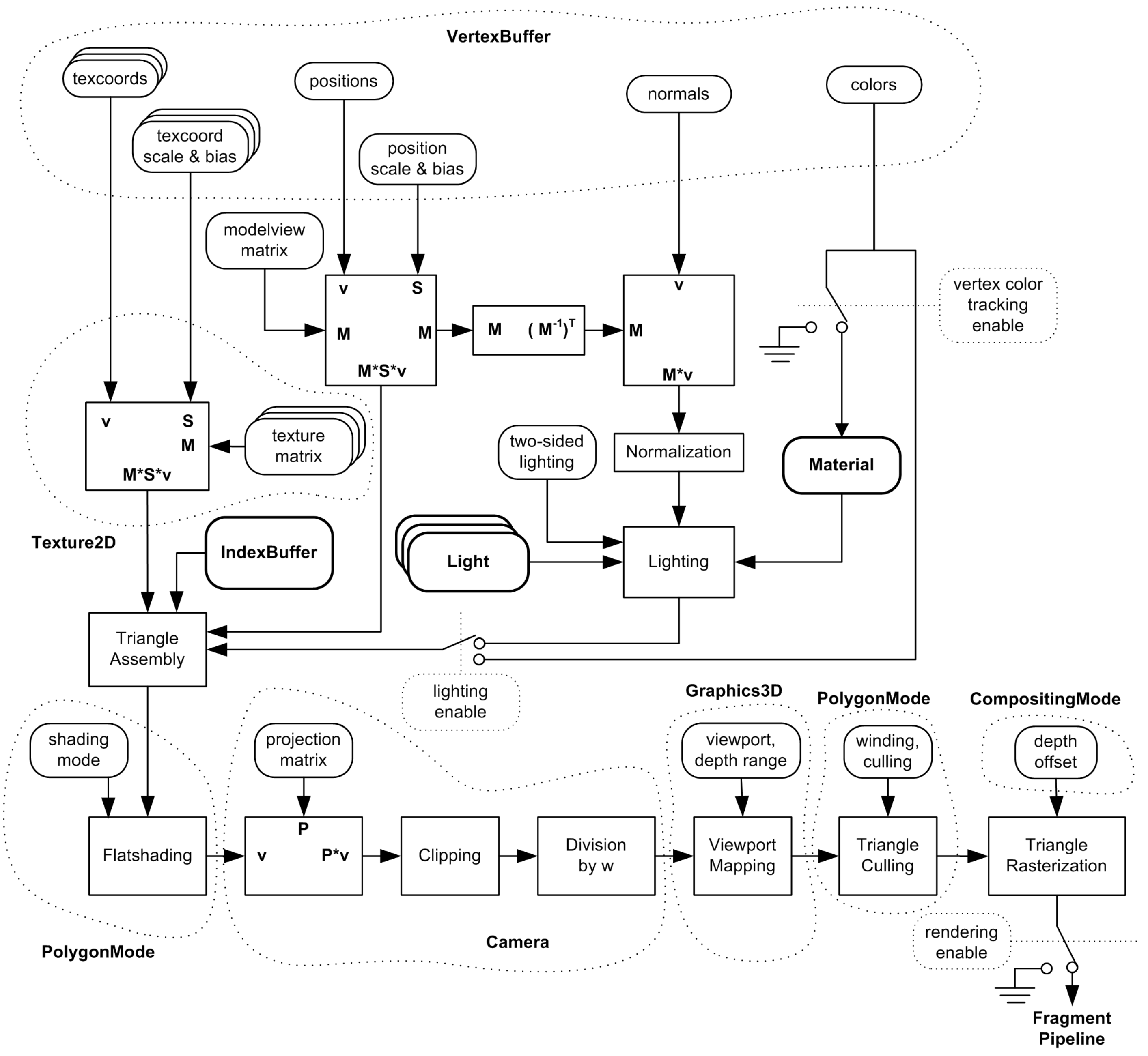

Grey objects are "Nodes" - groups or children of groups
White objects are components
Component
Arrows indicate references. These components are referenced by the node.
Node
Component
Component
Components can reference other components, and nodes can reference other nodes.

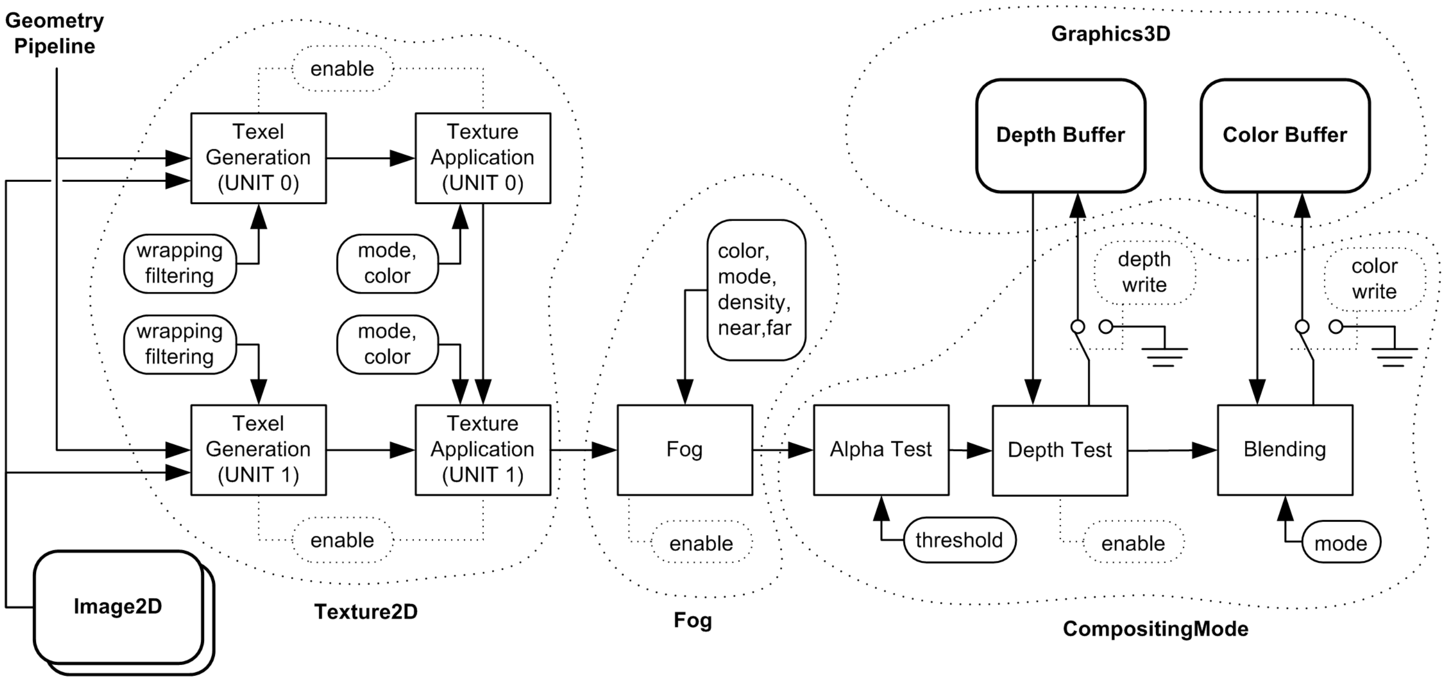
# General Implementation Requirements

## Rasterization

By default, vertices, indices, triangles, and fragments are processed as in OpenGL. In particular, triangle rasterization is done as specified in section 3.5.1 of the OpenGL specification.

The reference geometry and fragment pipelines are shown below. A rough mapping of Mesh components and other objects to the pipeline stages is also shown. Note that the ordering of the stages is the same as in OpenGL. Implementations may optimize their operation by doing things in a different order, but only if the result is exactly the same as it would be with the reference pipelines.


VertexBuffer
texcoords
texcoord scale & bias
positions
position scale & bias
normals
colors
modelview matrix
v
S
M
M
M*S*v
M
( M$^{-1}$)$^{T}$
M
v
M*v
vertex color tracking enable
Normalization
Material
v
S
M
M*S*v
texture matrix
two-sided lighting
Texture2D
IndexBuffer
Light
Lighting
Triangle Assembly
lighting enable
Graphics3D
PolygonMode
CompositingMode
shading mode
projection matrix
viewport, depth range
winding, culling
depth offset
Flatshading
P
v
P*v
Clipping
Division by w
Viewport Mapping
Triangle Culling
Triangle Rasterization
rendering enable
PolygonMode
Camera
Fragment Pipeline


Geometry Pipeline
enable
Texel Generation (UNIT 0)
Texture Application (UNIT 0)
wrapping filtering
mode, color
wrapping filtering
mode, color
Texel Generation (UNIT 1)
Texture Application (UNIT 1)
enable
Image2D
Texture2D
Graphics3D
Depth Buffer
Color Buffer
color, mode, density, near,far
depth write
color write
Fog
Alpha Test
Depth Test
Blending
enable
threshold
enable
mode
Fog
CompositingMode

## Numeric Range and Accuracy

The floating point format used for input and output is the standard IEEE float, having an 8-bit exponent and a 24-bit mantissa normalized to [1.0, 2.0). To facilitate efficient operation without floating point hardware, implementations are allowed to substitute more constrained representations internally. The internal format, and conversion from the input format to the internal format, must satisfy the following:

- The *numeric range* must be at least $R = [-2^{63}, 2^{63}]$. Values outside of R may produce undefined results.
- The *minimum absolute value* must be at most $d = 2^{-63}$. Smaller absolute values may be flushed to zero.
- The *precision* must be at least p = 16 significant bits for all values x in R, abs(x) >= d.

These requirements also apply to elementary arithmetic operations, which include addition, subtraction and multiplication. The operands are then taken to be in the internal format rather than the input format, and the value against which the precision is measured is taken to be the mathematically correct result, rounded to the nearest representable value. In addition, elementary arithmetic operations must satisfy the following:

- $x \cdot 0 = 0 \cdot x = 0$, for all values x in R.
- $1 \cdot x = x \cdot 1 = x$, for all values x in R.
- $x + 0 = 0 + x = x$, for all values x in R.
- $0^0 = 1$.

These requirements apply to all operations in this API, except rasterization and per-fragment operations, such as depth buffering, blending, and interpolation of colors and texture coordinates. In particular, the requirements do apply to node transformations in the scene graph; vertex coordinate, texture coordinate and normal vector transformations; picking; keyframe interpolation; mesh morphing; skinning; and all methods in the Transform class.

Blending, interpolation, comparisons and other operations on color, alpha and (screen-space) depth values must have a numeric range, minimum absolute value, and precision at least equivalent to the corresponding channel in the frame buffer. For example, an 8-bit color channel has R = [0, 1], d = 1/255, and p = 8. Within that domain, the rules are as specified above, with two additional requirements:

- All operations must be done component-wise and clamped to [0, 1].
- $a \cdot s + (1- a) \cdot s = s$, for all values a and s in [0, 1].

Loss of precision is allowed when converting the result of the operation into the frame buffer format, which is commonly fixed-point. The higher-precision internal value may be rounded to either of the two closest representable values in the frame buffer format. Note that the final precision will get progressively worse as the intermediate result approaches zero. In the worst case, all significant bits except the leading zero or one will be lost.

## Correspondence of Getters and Setters

When querying the value of some property in the API, the returned value does not need to be exactly the same that was set with the corresponding `set` method. Instead, it may be any value that *produces an equivalent result*. The returned values are also not required to be in any "canonical" or "normalized" form. In the case of node orientation, for example, there are a number of different axis-angle combinations that specify the same orientation, and any one of them may be returned.

The returned value may also be an approximation of the original value, as long as the accuracy constraints for the particular type of data are satisfied.

## References to Objects

Object3D instances are always held by reference rather than copied in. Changes to an Object3D therefore have immediate effect in any referring Object3D. For example, changes to an Image2D attached to a Background take effect without having to call the `Background.setImage` method again.

Objects that are not instances of Object3D are copied in by default. Any exceptions to this rule are clearly documented in the individual method descriptions. Note that arrays are Objects in Java, and are therefore copied in rather than held by reference. Also note that the Transform class, although defined in this API, is not derived from Object3D.

To clarify the handling of arrays, consider a hypothetical class X that takes in an Object3D array in its constructor. The constructor copies in the array, but stores the *elements* of the array by reference. Thus, replacing one Object3D in the array with another will have no effect on the instance of X that was just created. Indeed, the application may freely reuse the array or leave it for garbage collection. By contrast, any modifications to the actual Object3D instances that were contained in the array will automatically be reflected in the new instance of X.

## Deferred exceptions

The scene graph as well as individual objects are allowed to remain in an incomplete or invalid state for as long as their contents are not actually needed by the implementation (for rendering or some other purpose). An IllegalStateException is thrown only when the objects really must be valid. This kind of deferred error checking is necessary for aggregate objects, whose validity depends on other objects that the application can add, remove or change at any time. There are four operations in this API that can throw these *deferred exceptions*: the `render` methods in Graphics3D, the `pick` methods in Group, the `align` method in Node, and the `animate` method in Object3D.

The fact that deferred exceptions may or may not be thrown, depending on whether the implementation actually needs the offending data, can cause varying behavior between different implementations. For example, some implementations may use visibility culling to remove objects from further processing without having to check their vertex arrays, while others may use a brute-force approach and push all objects through the rendering pipeline. To reduce this variability without restricting innovation, implementations must obey the following rules when rendering or picking:

1. Objects that are out of scope or disabled **must not** be validated.
2. Objects that are not rendered or picked, even though they are enabled and within scope, **may** be validated.
3. Any data that are required in order for rendering or picking to produce meaningful results **must** be validated.

A Node can be disabled by clearing its rendering and picking enable flags. A submesh can be disabled by setting its Appearance to null. By definition, all objects are disabled when rendering from a Camera that has zero view volume.

## Thread Safety

Implementations must not crash or throw an exception as a result of being accessed from multiple threads at the same time. However, the results of the requested operation in that case may be unpredictable.

No method in this API is allowed to block waiting for a resource, such as a rendering target, to be released. This is to guarantee that no deadlock situations will occur. Also, any resources required by a method must be released upon return. No method is allowed to leave its host object or other resources locked.

## Pixel Format Conversion

Several different pixel formats are supported in rendering targets, textures, sprites, and background images. Depending on the case, a mismatch between the source and destination pixel formats may require a format conversion to be done. The general rules that are obeyed throughout the API are as follows:

- Luminance to RGB: The luminance value is replicated to each of R, G and B.
- Luminance to Alpha: The luminance value is copied in as the alpha value.
- RGBA to Alpha: The alpha value is copied in, and the RGB values are discarded.
- RGB to Alpha: Unspecified, but must take all components into account. For example, (R+G+B) / 3.
- RGB to Luminance: Unspecified, but must take all components into account. For example, (R+G+B) / 3.
- Alpha to Luminance: The alpha value is copied in as the luminance value.
- Any missing luminance, color or alpha components are set to 1.0, unless explicitly stated otherwise.

More specific rules related to pixel formats are specified on a case-by-case basis in classes dealing with images and the frame buffer. These include Graphics3D, Image2D, Texture2D, CompositingMode and Background.

## Example Applications

Two example MIDlets using the API are presented below. The first MIDlet is a pure immediate mode application that displays a rotating, texture-mapped cube. It shows how to initialize a 3D graphics context, bind it to a MIDP Canvas, and render some simple content with it. It also illustrates how to create a Mesh object "manually", that is, how to set up the coordinates, triangle connectivity, texture maps, and materials. In practice, this is usually not done programmatically, but with a 3D modeling tool. Loading a ready-made Mesh object with all the necessary attributes is a simple matter of calling the `load` method in Loader.

The other example MIDlet is a retained mode application that plays back a ready-made animation that it downloads over http.

**Examples:**
**(1) Immediate mode example MIDlet: Class MyCanvas.**

```
import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

public class MyCanvas extends Canvas {

    private Graphics3D      iG3D;
    private Camera          iCamera;
    private Light           iLight;
    private float           iAngle = 0.0f;
    private Transform       iTransform = new Transform();
    private Background      iBackground = new Background();
    private VertexBuffer    iVb;    // positions, normals, colors, texcoords
    private IndexBuffer     iIb;    // indices to iVB, forming triangle strips
    private Appearance      iAppearance; // material, texture, compositing, ...
    private Material        iMaterial = new Material();
    private Image           iImage;

    /**
     * Construct the Displayable.
     */
    public MyCanvas() {
        // set up this Displayable to listen to command events
        setCommandListener(new CommandListener()  {
            public void commandAction(Command c, Displayable d) {
                if (c.getCommandType() == Command.EXIT) {
                    // exit the MIDlet
                    MIDletMain.quitApp();
                }
            }
        });
        try {
            init();
        }
        catch(Exception e) {
             e.printStackTrace();
        }
    }

    /**
     * Component initialization.
     */
    private void init() throws Exception  {
        // add the Exit command
        addCommand(new Command("Exit", Command.EXIT, 1));

        // get the singleton Graphics3D instance
        iG3D = Graphics3D.getInstance();

        // create a camera
        iCamera = new Camera();
        iCamera.setPerspective( 60.0f,              // field of view
            (float)getWidth()/ (float)getHeight(),  // aspectRatio
            1.0f,      // near clipping plane
            1000.0f ); // far clipping plane

        // create a light
        iLight = new Light();
        iLight.setColor(0xffffff);         // white light
        iLight.setIntensity(1.25f);          // overbright

        // init some arrays for our object (cube)

        // Each line in this array declaration represents a triangle strip for
        // one side of a cube. The only primitive we can draw with is the
        // triangle strip so if we want to make a cube with hard edges we
        // need to construct one triangle strip per face of the cube.
        // 1 * * * * * 0
        //   * *     *
        //   *   *   *
        //   *     * *
        // 3 * * * * * 2
        // The ascii diagram above represents the vertices in the first line
        // (the first tri-strip)
        short[] vert = {
            10, 10, 10,  -10, 10, 10,   10,-10, 10,  -10,-10, 10,   // front
           -10, 10,-10,   10, 10,-10,  -10,-10,-10,   10,-10,-10,   // back
           -10, 10, 10,  -10, 10,-10,  -10,-10, 10,  -10,-10,-10,   // left
            10, 10,-10,   10, 10, 10,   10,-10,-10,   10,-10, 10,   // right
            10, 10,-10,  -10, 10,-10,   10, 10, 10,  -10, 10, 10,   // top
            10,-10, 10,  -10,-10, 10,   10,-10,-10,  -10,-10,-10 }; // bottom

        // create a VertexArray to hold the vertices for the object
        VertexArray vertArray = new VertexArray(vert.length / 3, 3, 2);
        vertArray.set(0, vert.length/3, vert);

        // The per-vertex normals for the cube; these match with the vertices
        // above. Each normal is perpendicular to the surface of the object at
        // the corresponding vertex.
        byte[] norm = {
            0, 0, 127,     0, 0, 127,     0, 0, 127,     0, 0, 127,
            0, 0,-127,     0, 0,-127,     0, 0,-127,     0, 0,-127,
           -127, 0, 0,    -127, 0, 0,    -127, 0, 0,    -127, 0, 0,
            127, 0, 0,     127, 0, 0,     127, 0, 0,     127, 0, 0,
            0, 127, 0,     0, 127, 0,     0, 127, 0,     0, 127, 0,
            0,-127, 0,     0,-127, 0,     0,-127, 0,     0,-127, 0 };

        // create a vertex array for the normals of the object
        VertexArray normArray = new VertexArray(norm.length / 3, 3, 1);
        normArray.set(0, norm.length/3, norm);

        // per vertex texture coordinates
        short[] tex = {
            1, 0,       0, 0,       1, 1,       0, 1,
            1, 0,       0, 0,       1, 1,       0, 1,
            1, 0,       0, 0,       1, 1,       0, 1,
            1, 0,       0, 0,       1, 1,       0, 1,
            1, 0,       0, 0,       1, 1,       0, 1,
            1, 0,       0, 0,       1, 1,       0, 1 };

        // create a vertex array for the texture coordinates of the object
        VertexArray texArray = new VertexArray(tex.length / 2, 2, 2);
        texArray.set(0, tex.length/2, tex);

        // the length of each triangle strip
        int[] stripLen = { 4, 4, 4, 4, 4, 4 };

        // create the VertexBuffer for our object
        VertexBuffer vb = iVb = new VertexBuffer();
        vb.setPositions(vertArray, 1.0f, null);      // unit scale, zero bias
        vb.setNormals(normArray);
        vb.setTexCoords(0, texArray, 1.0f, null);    // unit scale, zero bias

        // create the index buffer for our object (this tells how to
        // create triangle strips from the contents of the vertex buffer).
        iIb = new TriangleStripArray( 0, stripLen );

        // load the image for the texture
        iImage = Image.createImage( "/texture.png" );

        // create the Image2D (we need this so we can make a Texture2D)
        Image2D image2D = new Image2D( Image2D.RGB, iImage );

        // create the Texture2D and enable mipmapping
        // texture color is to be modulated with the lit material color
        Texture2D texture = new Texture2D( image2D );
        texture.setFiltering(Texture2D.FILTER_NEAREST,
                             Texture2D.FILTER_NEAREST);
        texture.setWrapping(Texture2D.WRAP_CLAMP,
                            Texture2D.WRAP_CLAMP);
        texture.setBlending(Texture2D.FUNC_MODULATE);

        // create the appearance
        iAppearance = new Appearance();
        iAppearance.setTexture(0, texture);
        iAppearance.setMaterial(iMaterial);
        iMaterial.setColor(Material.DIFFUSE, 0xFFFFFFFF);   // white
        iMaterial.setColor(Material.SPECULAR, 0xFFFFFFFF);  // white
        iMaterial.setShininess(100.0f);

        iBackground.setColor(0xf54588); // set the background color
```

```
    }

    /**
     * Paint the scene.
     */
    protected void paint(Graphics g) {

        // Bind the Graphics of this Canvas to our Graphics3D. The
        // viewport is automatically set to cover the entire clipping
        // rectangle of the Graphics object. The parameters indicate
        // that z-buffering, dithering and true color rendering are
        // enabled, but antialiasing is disabled.

        iG3D.bindTarget(g, true,
                        Graphics3D.DITHER |
                        Graphics3D.TRUE_COLOR);

        // clear the color and depth buffers
        iG3D.clear(iBackground);

        // set up the camera in the desired position
        Transform transform = new Transform();
        transform.postTranslate(0.0f, 0.0f, 30.0f);
        iG3D.setCamera(iCamera, transform);

        // set up a "headlight": a directional light shining
        // from the direction of the camera
        iG3D.resetLights();
        iG3D.addLight(iLight, transform);

        // update our transform (this will give us a rotating cube)
        iAngle += 1.0f;
        iTransform.setIdentity();
        iTransform.postRotate(iAngle,       // rotate 1 degree per frame
                       1.0f, 1.0f, 1.0f);  // rotate around this axis

        // Render our cube. We provide the vertex and index buffers
        // to specify the geometry; the appearance so we know what
        // material and texture to use; and the transform to tell
        // where to render the object
        iG3D.render(iVb, iIb, iAppearance, iTransform);

        // flush
        iG3D.releaseTarget();
    }
}
```

**(2) Immediate mode example MIDlet: Class MIDletMain.**

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class MIDletMain extends MIDlet
{
    static MIDletMain instance;
    MyCanvas displayable = new MyCanvas();
    Timer iTimer = new Timer();

    /**
```

```
      * Construct the midlet.
      */
     public MIDletMain() {
         this.instance = this;
     }

     /**
      * Main method.
      */
     public void startApp() {
         Display.getDisplay(this).setCurrent(displayable);
         iTimer.schedule( new MyTimerTask(), 0, 40 );
     }

     /**
      * Handle pausing the MIDlet.
      */
     public void pauseApp() {
     }

     /**
      * Handle destroying the MIDlet.
      */
     public void destroyApp(boolean unconditional) {
     }

     /**
      * Quit the MIDlet.
      */
     public static void quitApp() {
         instance.destroyApp(true);
         instance.notifyDestroyed();
         instance = null;
     }

     /**
      * Our timer task for providing animation.
      */
     class MyTimerTask extends TimerTask {
         public void run() {
             if( displayable != null ) {
                 displayable.repaint();
             }
         }
     }
}
```

**(3) Retained mode example MIDlet.**

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.CommandListener;

import java.util.Timer;
```

```java
import java.util.TimerTask;

import javax.microedition.m3g.*;

public class JesterTestlet extends MIDlet implements CommandListener
{
    private Display myDisplay = null;
    private JesterCanvas myCanvas = null;

    private Timer myRefreshTimer = new Timer();
    private TimerTask myRefreshTask = null;

    private Command exitCommand = new Command("Exit", Command.ITEM, 1);

    private World myWorld = null;

    /**
     * JesterTestlet - default constructor.
     */
    public JesterTestlet()
    {
        // Set up the user interface.
        myDisplay = Display.getDisplay(this);
        myCanvas = new JesterCanvas(this);
        myCanvas.setCommandListener(this);
        myCanvas.addCommand(exitCommand);
    }

    /**
     * startApp()
     */
    public void startApp() throws MIDletStateChangeException
    {
        myDisplay.setCurrent(myCanvas);

        try
        {
            // Load a file.
            Object3D[] roots =
                Loader.load("http://www.example.com/m3g/samples/simple.m3g");

            // Assume the world is the first root node loaded.
            myWorld = (World)roots[0];

            // Force a repaint so that we get the update loop started.
            myCanvas.repaint();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    /**
     * pauseApp()
     */
    public void pauseApp()
    {
        // Release resources.
        myWorld = null;
    }
    /**
     * destroyApp()
     */
    public void destroyApp(boolean unconditional) throws MIDletStateChangeException
    {
        myRefreshTimer.cancel();
        myRefreshTimer = null;

        // Release resources.
        myWorld = null;
    }

    /**
     * MIDlet paint method.
     */
    public void paint(Graphics g)
    {
        // We are not fully initialised yet; just return.
        if(myCanvas == null || myWorld == null)
            return;

        // Delete any pending refresh tasks.
        if(myRefreshTask != null)
        {
            myRefreshTask.cancel();
            myRefreshTask = null;
        }

        // Get the current time.
        long currentTime = System.currentTimeMillis();
        // Update the world to the current time.
        int validity = myWorld.animate((int)currentTime);

        // Render to our Graphics.
        Graphics3D myGraphics3D = Graphics3D.getInstance();
        myGraphics3D.bindTarget(g);
        myGraphics3D.render(myWorld);
        myGraphics3D.releaseTarget();

        // Subtract time taken to do the update.
        validity -= System.currentTimeMillis() - currentTime;

        if(validity < 1)
        {    // The validity is too small; allow a minimum of 1ms.
            validity = 1;
        }

        // If the validity is not infinite schedule a refresh task.
        if(validity < 0x7fffffff)
        {
            // Create a new refresh task.
            myRefreshTask = new RefreshTask();

            // Schedule an update.
            myRefreshTimer.schedule(myRefreshTask, validity);
        }
    }

    /**
     * Handle commands.
     */
    public void commandAction(Command cmd, Displayable disp)
    {
        if (cmd == exitCommand)
        {
            try
            {
                destroyApp(false);
                notifyDestroyed();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }

    /**
     * Inner class for refreshing the view.
     */
    private class RefreshTask extends TimerTask
    {
        public void run()
        {
            // Get the canvas to repaint itself.
            myCanvas.repaint();
        }
    }

    /**
     * Inner class for handling the canvas.
     */
    class JesterCanvas extends Canvas
    {
        JesterTestlet myTestlet;

        /**
         * Construct a new canvas
         */
        JesterCanvas(JesterTestlet Testlet) { myTestlet = Testlet; }

        /**
         * Initialize self.
         */
        void init() { }

        /**
         * Cleanup and destroy.
         */
        void destroy() { }

        /**
         * Ask myTestlet to paint itself
         */
        protected void paint(Graphics g) { myTestlet.paint(g); }
    }
}
```

# Class Hierarchy

- class java.lang.Object
  - class javax.microedition.m3g.**Graphics3D**
  - class javax.microedition.m3g.**Loader**
  - class javax.microedition.m3g.**Object3D**
    - class javax.microedition.m3g.**AnimationController**
    - class javax.microedition.m3g.**AnimationTrack**
    - class javax.microedition.m3g.**Appearance**
    - class javax.microedition.m3g.**Background**
    - class javax.microedition.m3g.**CompositingMode**
    - class javax.microedition.m3g.**Fog**
    - class javax.microedition.m3g.**Image2D**
    - class javax.microedition.m3g.**IndexBuffer**
      - class javax.microedition.m3g.**TriangleStripArray**
    - class javax.microedition.m3g.**KeyframeSequence**
    - class javax.microedition.m3g.**Material**
    - class javax.microedition.m3g.**PolygonMode**
    - class javax.microedition.m3g.**Transformable**
      - class javax.microedition.m3g.**Node**
        - class javax.microedition.m3g.**Camera**
        - class javax.microedition.m3g.**Group**
          - class javax.microedition.m3g.**World**
        - class javax.microedition.m3g.**Light**
        - class javax.microedition.m3g.**Mesh**
          - class javax.microedition.m3g.**MorphingMesh**
          - class javax.microedition.m3g.**SkinnedMesh**
        - class javax.microedition.m3g.**Sprite3D**
      - class javax.microedition.m3g.**Texture2D**
    - class javax.microedition.m3g.**VertexArray**
    - class javax.microedition.m3g.**VertexBuffer**
  - class javax.microedition.m3g.**RayIntersection**
  - class javax.microedition.m3g.**Transform**

**javax.microedition.m3g**

# Class AnimationController

```
java.lang.Object
  └─ javax.microedition.m3g.Object3D
       └─ javax.microedition.m3g.AnimationController
```

public class **AnimationController**
extends Object3D

Controls the position, speed and weight of an animation sequence.

In anything other than the simplest scenes, an animation sequence will require control of more than one property of more than one object. For example, a jointed figure performing a single gesture is usually thought of as a single animation, yet it involves the coordinated control of the position and orientation of many different objects.

We define an *animation sequence* to mean a set of individual AnimationTracks that are controlled by a single AnimationController. Each AnimationTrack object contains all the data required to control a single animatable property on one target object.

An AnimationController object enables its associated animation sequence as a whole to be paused, stopped, restarted, fast-forwarded, rewound, arbitrarily repositioned, or deactivated. More formally, it defines a linear mapping from world time to sequence time.

The detailed behaviour of how the data flows through the animation system as a whole is documented in the `AnimationTrack` class.

## Animation application

In both immediate and retained mode, animations are explicitly applied to target objects by calling the `animate` method on the target Object3D itself. This re-evaluates the values of all object properties that have one or more animations attached. Animations are also applied to the children of the target object, so the application is free to choose between calling `myWorld.animate` to animate everything in `myWorld` at once, or applying animations to more fine-grained groups of objects individually.

Animation controllers have an active interval, specified by minimum and maximum world time values, during which the animation controller is active. Animations controlled by inactive animation controllers have no effect on their target objects and are simply ignored during animation application.

## Animation weighting

Each animation controller has a *weight* associated with it. The contributions of all active animations targeting the same property at the same time are blended together by their respective weights. Formally, the value of a scalar property P as a function of weights $w_i$ and contributions $P_i$ is:

$$P = \text{sum} [ w_i P_i ]$$

For vector-valued properties, the above formula is applied for each vector component separately.

For most types of animation, the simple weighted sum as shown above is sufficient, but for orientation values the implementation is required to normalize the resulting quaternion. The quaternion must be normalized even if there is only one active animation controller, and that controller has unit weight, so that no actual weighting or blending takes place. Note also that the individual contributing quaternions $P_i$ must *not* be normalized prior to weighting.

## Timing and speed control

AnimationController specifies a linear mapping between *world time*, passed in to `Object3D.animate`, and *sequence time* that is used in sampling the associated keyframe data.

The sequence time is calculated directly from the given world time at each call to `animate`, instead of storing it internally. This is to avoid undesirable accumulation of rounding errors and any artifacts potentially resulting from that. It also simplifies the usage of the animation system by making it effectively stateless (as opposed to a traditional state machine design).

The mapping from world time to sequence time is parameterized by three constants, specified in AnimationController, and one variable, the world time, that is passed in to `animate`. The formula for calculating the sequence time $t_s$ corresponding to a given world time $t_w$ is:

$$t_s = t_{sref} + s\,(t_w - t_{wref})$$

where

$t_s$ = the computed sequence time
$t_w$ = the given world time
$t_{sref}$ = the reference sequence time
$t_{wref}$ = the reference world time
s = the speed; sequence time per world time

The *reference point* $(t_{wref}, t_{sref})$ is specified with the `setPosition` method and the speed with the `setSpeed` method (note that setting the speed may also change the reference point).

Sequence time can be visualized, in a coordinate system where world time is on the horizontal and sequence time on the vertical axis, as a line having slope s and passing through the point $(t_{wref}, t_{sref})$.

As an example of the relationship between world time and sequence time, imagine a world where the current time is 5000 milliseconds since the start. An animation was started (from 0 ms sequence time) at 3000 ms, running at half speed. The animation was started 2000 ms ago, but because the speed is 0.5, the actual required sequence time $t_{sref}$ is 1000 ms. Here, we would have $t_w$ = 5000 ms, $t_{wref}$ = 3000 ms, $t_{sref}$ = 0 ms, and s = 0.5 in the formula above.

Note that the unit of time is not explicitly specified anywhere in the API or the file format. It is strongly recommended that applications and content creation tools express times in milliseconds by default. Arbitrary units can, however, be used in specific applications if mandated by range or precision requirements.

## Synchronized animation

We assume that synchronization of animation with other media types is only based on the world time passed from the controlling application. No synchronization events or other mechanisms are provided for this purpose. In the case of synchronizing animation to music, for example, the current elapsed time is often available directly from the music player library.

## Example usage

As an example of using animation, consider a case where we want a light source to pulsate between red and green, moving along a curved path. In both immediate and retained mode, this involves creating keyframe sequences and associating them with the light node, as illustrated in Example 1 below.

To apply the animation to the light object in our rendering loop, we must call the `animate` method, as shown in Example 2.

**See Also:**

Binary format, `AnimationTrack`, `KeyframeSequence`, `Object3D`

**Examples:**

**(1) Creating an animation.**

```
Light light = new Light();      // Create a light node

// Load a motion path from a stream, assuming it's the first
// object there

Object3D[] objects = Loader.load("http://www.ex.com/ex.m3g");
KeyframeSequence motion = (KeyframeSequence) objects[0];

// Create a color keyframe sequence, with keyframes at 0 ms
// and 500 ms, and a total duration of 1000 ms. The animate
// method will throw an exception if it encounters a
// KeyframeSequence whose duration has not been set or whose
// keyframes are out of order. Note that the Loader
// automatically validates any sequences that are loaded from
// a file.

KeyframeSequence blinking = new KeyframeSequence(2, 3,
                                  KeyframeSequence.LINEAR);
blinking.setKeyframe(0,   0, new float[] { 1.0f, 0.0f, 0.0f });
blinking.setKeyframe(1, 500, new float[] { 0.0f, 1.0f, 0.0f });
blinking.setDuration(1000);

AnimationTrack blink = new AnimationTrack(blinking,
                                    AnimationTrack.COLOR);
AnimationTrack move = new AnimationTrack(motion,
                              AnimationTrack.TRANSLATION);
light.addAnimationTrack(blink);
light.addAnimationTrack(move);

// Create an AnimationController and make it control both the
// blinking and the movement of our light

AnimationController lightAnim = new AnimationController();
blink.setController(lightAnim);
```

```
move.setController(lightAnim);

// Start the animation when world time reaches 2 seconds, stop
// at 5 s.  There is only one reference point for this
// animation: sequence time must be zero at world time 2000
// ms. The animation will be running at normal speed (1.0, the
// default).

lightAnim.setActiveInterval(2000, 5000);
lightAnim.setPosition(0, 2000);
```

**(2) Applying the animation during rendering.**

```
appTime += 30;           // advance time by 30 ms each frame
light.animate(appTime);

// Assume 'myGraphics3D' is the Graphics3D object we draw into.
// In immediate mode, node transforms are ignored, so we get
// our animated transformation into a local Transform object,
// "lightToWorld". As its name implies, the transformation is
// from the Light node's local coordinates to world space.

light.getTransform(lightToWorld);
myGraphics3D.resetLights();
myGraphics3D.addLight(light, lightToWorld);
```

## Constructor Summary

| |
|---|
| `AnimationController()`<br>Creates a new AnimationController object. |

## Method Summary

| | |
|---|---|
| `int` | `getActiveIntervalEnd()`<br>Retrieves the ending time of the current active interval of this animation controller, in world time units. |
| `int` | `getActiveIntervalStart()`<br>Retrieves the starting time of the current active interval of this animation controller, in world time units. |
| `float` | `getPosition(int worldTime)`<br>Retrieves the sequence time that corresponds to the given world time. |
| `int` | `getRefWorldTime()`<br>Returns the current reference world time. |
| `float` | `getSpeed()`<br>Retrieves the currently set playback speed of this animation controller. |
| `float` | `getWeight()`<br>Retrieves the currently set blending weight for this animation controller. |
| `void` | `setActiveInterval(int start, int end)`<br>Sets the world time interval during which this animation controller is active. |

| | |
|---|---|
| void | `setPosition(float sequenceTime, int worldTime)` Sets a new playback position, relative to world time, for this animation controller. |
| void | `setSpeed(float speed, int worldTime)` Sets a new playback speed for this animation. |
| void | `setWeight(float weight)` Sets the blending weight for this animation controller. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### AnimationController

```
public AnimationController()
```

Creates a new AnimationController object. The default values for the new object are:

- active interval: [0, 0) (always active)
- blending weight: 1.0
- speed: 1.0
- reference point: (0, 0)

## Method Detail

### setActiveInterval

```
public void setActiveInterval(int start,
                              int end)
```

Sets the world time interval during which this animation controller is active.

This animation controller will subsequently be *active* when the world time t is such that `start <= t < end`, and inactive outside of that range. As a special case, if `start` and `end` are set to the same value, this animation controller is always active.

Note that changing the active interval has no effect on the mapping from world time to sequence time.

**Parameters:**
`start` - the starting time of the active interval, in world time units (inclusive)
`end` - the ending time of the active interval, in world time units (exclusive)
**Throws:**

`java.lang.IllegalArgumentException` - if `start > end`

**See Also:**

`getActiveIntervalStart`, `getActiveIntervalEnd`

## getActiveIntervalStart

```
public int getActiveIntervalStart()
```

Retrieves the starting time of the current active interval of this animation controller, in world time units. The value returned is the same that was last set with `setActiveInterval`, or if it has not been called yet, the default value set at construction.

**Returns:**

the starting time of the active interval

**See Also:**

`setActiveInterval`

## getActiveIntervalEnd

```
public int getActiveIntervalEnd()
```

Retrieves the ending time of the current active interval of this animation controller, in world time units. The value returned is the same that was last set with `setActiveInterval`, or if it has not been called yet, the default value set at construction.

**Returns:**

the ending time of the active interval

**See Also:**

`setActiveInterval`

## setSpeed

```
public void setSpeed(float speed,
                     int worldTime)
```

Sets a new playback speed for this animation. The speed is set as a factor of the nominal speed of the animation: 1.0 is normal playback speed (as specified by the keyframe times in the associated animation tracks), 2.0 is double speed, and -1.0 is reverse playback at normal speed. A speed of 0.0 freezes the animation.

The speed setting effectively specifies how much to advance the internal playback position of this animation for a given increment in the global world time.

The internal reference point is modified so that sequence time at the given world time remains unchanged. This allows the application to change the speed without causing the animation to "jump" forward or backward. To get the desired effect, the application should pass its current world time to this method. This is the time that the application has most recently used in `animate`, or the time that it is next going to use.

The reference point ($t_{wref}$, $t_{sref}$) and speed (s) are updated based on the given world time and speed as follows:

$t_{wref}$' = worldTime
$t_{sref}$' = getPosition(worldTime)
s' = speed

Note that the computation of the new reference sequence time takes place before updating the speed. See the class description for the formula that `getPosition` uses, and for more discussion on animation timing.

**Parameters:**
`speed` - new playback speed; 1.0 is normal speed
`worldTime` - reference world time; the value of sequence time at this point will remain constant during the speed change
**See Also:**
`getSpeed`

## getSpeed

```
public float getSpeed()
```

Retrieves the currently set playback speed of this animation controller.

**Returns:**
the current playback speed
**See Also:**
`setSpeed`

## setPosition

```
public void setPosition(float sequenceTime,
                        int worldTime)
```

Sets a new playback position, relative to world time, for this animation controller. This sets the internal reference point ($t_{wref}$, $t_{sref}$) to (`worldTime`, `sequenceTime`) to shift the animation to the new position.

**Parameters:**
`sequenceTime` - the desired playback position in sequence time units
`worldTime` - the world time at which the sequence time must be equal to `sequenceTime`
**See Also:**
`getPosition`

## getPosition

```
public float getPosition(int worldTime)
```

Retrieves the sequence time that corresponds to the given world time. The returned value is computed with the formula given in the class description. Note that because the result may be a fractional number, it is returned as a float, not integer.

**Parameters:**
`worldTime` - world time to get the corresponding sequence time of

**Returns:**
animation sequence position in number of time units elapsed since the beginning of this animation, until `worldTime`

**See Also:**
`setPosition`

## getRefWorldTime

```
public int getRefWorldTime()
```

Returns the current reference world time.

**Returns:**
the current reference world time

**Since:**
M3G 1.1

**See Also:**
`setPosition`

## setWeight

```
public void setWeight(float weight)
```

Sets the blending weight for this animation controller. The blending weight must be positive or zero. Setting the weight to zero disables this animation controller; that is, the controller is subsequently not active even within its active range. If the weight is non-zero, the animations controlled by this controller contribute to their target properties as described in the class description.

**Parameters:**
`weight` - the new blending weight

**Throws:**
`java.lang.IllegalArgumentException` - if `weight < 0`

**See Also:**
`getWeight`

## getWeight

```
public float getWeight()
```

Retrieves the currently set blending weight for this animation controller.

**Returns:**
the current blending weight

**See Also:**
`setWeight`

**javax.microedition.m3g**

# Class AnimationTrack

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.AnimationTrack
```

public class **AnimationTrack**
extends Object3D

Associates a KeyframeSequence with an AnimationController and an animatable property.

An *animatable property* is a scalar or vector variable that the animation system can directly update; for instance, the orientation of a Node. Animatable properties are identified by the symbolic constants listed below. Some animatable properties are only applicable to one class, such as the `SHININESS` of a Material, while others apply to two or more classes.

Most classes derived from Object3D have one or more animatable properties. An Object3D instance with animatable properties is called an *animatable object*. Each animatable property of an animatable object constitutes a unique *animation target*.

Each animatable object may reference zero or more AnimationTracks. Each of these, when activated by their respective AnimationControllers, is in charge of updating one of the animation targets of the animatable object. The values assigned to the targets are determined by sampling the KeyframeSequence objects referenced by the AnimationTrack objects. Each KeyframeSequence can be referenced by multiple AnimationTracks, allowing the keyframe data to be shared.

Each AnimationTrack is associated with exactly one AnimationController, one KeyframeSequence, and one animatable property, but it may be associated with multiple animation targets. In other words, it can animate the same property in many different objects simultaneously. It is also possible to have several AnimationTrack objects associated with a single animation target. In this case, the final value of the animation target is a linear combination of the values derived from the individual AnimationTracks, weighted by their respective AnimationController weights.

## Implementation guidelines

### Clamping of interpolated values

Animation keyframes are input as floating point, and the values produced after interpolation and blending are also in floating point. When applied to their target property, the values must be mapped to the closest representable value that is valid for the property in question. For example, values for a floating point property must be clamped to the valid range for that property; values for an integer property rounded to the closest integer; and values for a boolean property interpreted as *true* when the value produced by animation is greater than or equal to 0.5, *false* if less. Exceptions to this rule are stated explicitly if necessary.

In summary, applying an animated quantity to its target property must never result in an exception or otherwise illegal state.

### Example implementation

When the whole scene graph or a subtree of objects is updated (using a call to `Object3D.animate`), the *world time*, maintained by the controlling application, is passed to each animatable object. In turn, each animatable object passes the world time to each of the AnimationTrack objects which are bound to it.

The AnimationTrack object then checks to see if the current world time falls within the active interval of its associated AnimationController object. If not, then no further action is taken by this AnimationTrack. If no active AnimationTrack objects are found for an animation target, the value of that target is unchanged. Note, however, that animation targets are independent of each other, and other targets in the same object may still change.

If the AnimationController is active, it is used to determine the *sequence time* for the animation. (Details of this calculation can be found in the AnimationController class description.) The sequence time is then used to obtain an interpolated value from the KeyframeSequence object. (Details of interpolation are in the KeyframeSequence class description.) This sample is then multiplied by the weight factor of the AnimationController object and applied to the target property. If multiple AnimationTrack objects target the same property, they are blended together according to the weights of their respective AnimationController objects; see AnimationController for more details on animation blending.

**See Also:**

Binary format, `KeyframeSequence`, `AnimationController`, `Object3D.addAnimationTrack`

## Field Summary

| | |
|---|---|
| `static int` | **`ALPHA`**<br>Specifies the alpha factor of a Node, or the alpha component of the Background color, Material diffuse color, or VertexBuffer default color as an animation target. |
| `static int` | **`AMBIENT_COLOR`**<br>Specifies the ambient color of a Material as an animation target. |
| `static int` | **`COLOR`**<br>Specifies the color of a Light, Background, or Fog, or the texture blend color in Texture2D, or the VertexBuffer default color as an animation target. |
| `static int` | **`CROP`**<br>Specifies the cropping parameters of a Sprite3D or Background as an animation target. |
| `static int` | **`DENSITY`**<br>Specifies the fog density in Fog as an animation target. |
| `static int` | **`DIFFUSE_COLOR`**<br>Specifies the diffuse color of a Material as an animation target. |
| `static int` | **`EMISSIVE_COLOR`**<br>Specifies the emissive color of a Material as an animation target. |
| `static int` | **`FAR_DISTANCE`**<br>Specifies the far distance of a Camera or Fog as an animation target. |
| `static int` | **`FIELD_OF_VIEW`**<br>Specifies the field of view of a Camera as an animation target. |
| `static int` | **`INTENSITY`**<br>Specifies the intensity of a Light as an animation target. |

| | |
|---|---|
| static int | **MORPH_WEIGHTS**<br>Specifies the morph target weights of a MorphingMesh as an animation target. |
| static int | **NEAR_DISTANCE**<br>Specifies the near distance of a Camera or Fog as an animation target. |
| static int | **ORIENTATION**<br>Specifies the orientation (R) component of a Transformable object as an animation target. |
| static int | **PICKABILITY**<br>Specifies the picking enable flag of a Node as an animation target. |
| static int | **SCALE**<br>Specifies the scale (S) component of a Transformable object as an animation target. |
| static int | **SHININESS**<br>Specifies the shininess of a Material as an animation target. |
| static int | **SPECULAR_COLOR**<br>Specifies the specular color of a Material as an animation target. |
| static int | **SPOT_ANGLE**<br>Specifies the spot angle of a Light as an animation target. |
| static int | **SPOT_EXPONENT**<br>Specifies the spot exponent of a Light as an animation target. |
| static int | **TRANSLATION**<br>Specifies the translation (T) component of a Transformable object as an animation target. |
| static int | **VISIBILITY**<br>Specifies the rendering enable flag of a Node as an animation target. |

## Constructor Summary

| |
|---|
| **AnimationTrack**(KeyframeSequence sequence, int property)<br>Creates an animation track with the given keyframe sequence targeting the given property. |

## Method Summary

| | |
|---|---|
| AnimationController | **getController**()<br>Retrieves the animation controller used for controlling this animation track. |
| KeyframeSequence | **getKeyframeSequence**()<br>Returns the keyframe sequence object which defines the keyframe values for this animation track. |
| int | **getTargetProperty**()<br>Returns the property targeted by this AnimationTrack. |
| void | **setController**(AnimationController controller)<br>Specifies the animation controller to be used for controlling this animation track. |

**Methods inherited from class javax.microedition.m3g.Object3D**

`addAnimationTrack`, `animate`, `duplicate`, `find`, `getAnimationTrack`, `getAnimationTrackCount`, `getReferences`, `getUserID`, `getUserObject`, `removeAnimationTrack`, `setUserID`, `setUserObject`

## Field Detail

### ALPHA

```
public static final int ALPHA
```

Specifies the alpha factor of a Node, or the alpha component of the Background color, Material diffuse color, or VertexBuffer default color as an animation target. The interpolated value is clamped to the range [0, 1].

Number of components required: 1

**See Also:**
Constant Field Values

### AMBIENT_COLOR

```
public static final int AMBIENT_COLOR
```

Specifies the ambient color of a Material as an animation target. The interpolated value of each color component is clamped to the range [0, 1].

Number of components required: 3 (RGB)

**See Also:**
Constant Field Values

### COLOR

```
public static final int COLOR
```

Specifies the color of a Light, Background, or Fog, or the texture blend color in Texture2D, or the VertexBuffer default color as an animation target. The interpolated value of each color component is clamped to the range [0, 1].

Note that the alpha component of the background color or default color is targeted separately using the identifier `ALPHA` (the other `COLOR` targets do not have an alpha component).

Number of components required: 3 (RGB)

**See Also:**
Constant Field Values

## CROP

```
public static final int CROP
```

Specifies the cropping parameters of a Sprite3D or Background as an animation target. The required parameters are the X and Y coordinates of the crop rectangle upper left corner, and the width and height of the crop rectangle, in that order.

The X and Y parameters may take on any value, regardless of whether the target object is a Sprite3D or Background. The width and height, however, have differing limits depending on the target.

In case of a Background target, negative values of width and height are clamped to zero. In case of a Sprite3D target, they are clamped to the range [-N, N], where N is the implementation specific maximum sprite crop size. Recall that negative values of width and height cause the displayed image to be flipped in the corresponding dimensions.

Number of components required: 2 (X, Y) or 4 (X, Y, width, height)

**See Also:**
Constant Field Values

## DENSITY

```
public static final int DENSITY
```

Specifies the fog density in Fog as an animation target. If the interpolated value is negative, it is clamped to zero.

Number of components required: 1

**See Also:**
Constant Field Values

## DIFFUSE_COLOR

```
public static final int DIFFUSE_COLOR
```

Specifies the diffuse color of a Material as an animation target. The interpolated value of each color component is clamped to the range [0, 1].

Note that the alpha component of the diffuse color is targeted separately, using the identifier ALPHA.

Number of components required: 3 (RGB)

**See Also:**
Constant Field Values

## EMISSIVE_COLOR

`public static final int` **`EMISSIVE_COLOR`**

Specifies the emissive color of a Material as an animation target. The interpolated values of the color components are clamped to the range [0, 1].

Number of components required: 3 (RGB)

**See Also:**
Constant Field Values

## FAR_DISTANCE

`public static final int` **`FAR_DISTANCE`**

Specifies the far distance of a Camera or Fog as an animation target. In case of a Camera target in perspective mode, negative values and zero are clamped to the smallest representable positive number. In case of a Fog target, or a camera target in parallel mode, the value is not clamped.

Animating any of the camera parameters (near, far, field of view) only has an effect if the camera is in perspective or parallel mode.

Number of components required: 1

**See Also:**
Constant Field Values

## FIELD_OF_VIEW

`public static final int` **`FIELD_OF_VIEW`**

Specifies the field of view of a Camera as an animation target. The interpolated value is clamped to the range (0, 180) in case of a perspective Camera. In case of a parallel camera, negative values and zero are clamped to the smallest representable positive number.

Animating any of the camera parameters (near, far, field of view) only has an effect if the camera is in perspective or parallel mode.

Number of components required: 1

**See Also:**
Constant Field Values

## INTENSITY

`public static final int` **`INTENSITY`**

Specifies the intensity of a Light as an animation target.

Number of components required: 1

**See Also:**
Constant Field Values

## MORPH_WEIGHTS

`public static final int` **`MORPH_WEIGHTS`**

Specifies the morph target weights of a MorphingMesh as an animation target. If there are N morph targets in the target mesh, the associated keyframes should be N-element vectors.

Since there is no direct reference from this object to its associated MorphingMesh node, there is no way to check at construction time that the number of vector components matches the number of morph targets. Denoting the number of components in the keyframe vectors by V, the following rules apply in case of a mismatch:

If V < N, then morph target weights are set as

w[i] = v[i], for 0 <= i <V
w[i] = 0.0, for V <= i <N

If V > N, then morph target weights are set as

w[i] = v[i], for 0 <= i <N
v[i] ignored for N <= i <V

Number of components required: N

**See Also:**
Constant Field Values

## NEAR_DISTANCE

`public static final int` **`NEAR_DISTANCE`**

Specifies the near distance of a Camera or Fog as an animation target. In case of a Camera target in perspective mode, negative values and zero are clamped to the smallest representable positive number. In case of a Fog target, or a camera target in parallel mode, the value is not clamped.

Animating any of the camera parameters (near, far, field of view) only has an effect if the camera is in perspective or parallel mode.

Number of components required: 1

**See Also:**
Constant Field Values

## ORIENTATION

`public static final int` **`ORIENTATION`**

Specifies the orientation (R) component of a Transformable object as an animation target.

The orientation is specified as a 4-element vector defining a quaternion. The quaternion components in the keyframes are ordered as follows:

- v[0] coefficient of i (related to the x component of the rotation axis)
- v[1] coefficient of j (related to the y component of the rotation axis)
- v[2] coefficient of k (related to the z component of the rotation axis)
- v[3] the scalar component (related to the rotation angle)

The quaternion resulting from interpolation is normalized automatically before applying it to the target, as specified in the AnimationController class description, section "Animation weighting".

Note that there are only two stages in the animation process where the implementation must normalize quaternions. They must not be normalized anywhere else. The first is when `SLERP` or `SQUAD` keyframes are fed in. The other is when the final, weighted result is applied to the target object.

Number of components required: 4

**See Also:**
Constant Field Values

## PICKABILITY

`public static final int` **`PICKABILITY`**

Specifies the picking enable flag of a Node as an animation target.

Number of components required: 1

**See Also:**
Constant Field Values

## SCALE

`public static final int` **`SCALE`**

Specifies the scale (S) component of a Transformable object as an animation target. The number of keyframe components in the associated KeyframeSequence can be either one or three, for uniform or non-uniform scaling, respectively.

Number of components required: 1 or 3 (XYZ)

**See Also:**

Constant Field Values

## SHININESS

```
public static final int SHININESS
```

Specifies the shininess of a Material as an animation target. The interpolated value is clamped to the range [0, 128].

Number of components required: 1

**See Also:**
Constant Field Values

## SPECULAR_COLOR

```
public static final int SPECULAR_COLOR
```

Specifies the specular color of a Material as an animation target. The interpolated value of each color component is clamped to the range [0, 1].

Number of components required: 3 (RGB)

**See Also:**
Constant Field Values

## SPOT_ANGLE

```
public static final int SPOT_ANGLE
```

Specifies the spot angle of a Light as an animation target. The interpolated value is clamped to the range [0, 90].

Number of components required: 1

**See Also:**
Constant Field Values

## SPOT_EXPONENT

```
public static final int SPOT_EXPONENT
```

Specifies the spot exponent of a Light as an animation target. The interpolated value is clamped to the range [0, 128].

Number of components required: 1

**See Also:**

Constant Field Values

### TRANSLATION

```
public static final int TRANSLATION
```

Specifies the translation (T) component of a Transformable object as an animation target.

Number of components required: 3 (XYZ)

**See Also:**
Constant Field Values

### VISIBILITY

```
public static final int VISIBILITY
```

Specifies the rendering enable flag of a Node as an animation target.

Number of components required: 1

**See Also:**
Constant Field Values

## Constructor Detail

### AnimationTrack

```
public AnimationTrack(KeyframeSequence sequence,
                      int property)
```

Creates an animation track with the given keyframe sequence targeting the given property. The keyframe sequence must be compatible with the target property; for example, to animate the translation component of a transformation, the keyframes must be 3-element vectors.

No controller is initially attached to the track.

**Parameters:**
`sequence` - a KeyframeSequence containing the keyframe data for this animation track
`property` - one of `ALPHA, ..., VISIBILITY`

**Throws:**
`java.lang.NullPointerException` - if `sequence` is null
`java.lang.IllegalArgumentException` - if `property` is not one of the symbolic constants listed above
`java.lang.IllegalArgumentException` - if `sequence` is not compatible with `property`

## Method Detail

### setController

`public void setController(AnimationController controller)`

Specifies the animation controller to be used for controlling this animation track. The controller determines the mapping from world time to sequence time, the speed of animation, and the active interval for all tracks under its control.

**Parameters:**

`controller` - an AnimationController object which defines the active state and sequence time for this animation sequence; if this is null then the behaviour is equivalent to associating this object with an inactive animation controller

**See Also:**

`getController`

### getController

`public AnimationController getController()`

Retrieves the animation controller used for controlling this animation track.

**Returns:**

the AnimationController object which defines the active state and sequence time for this animation sequence, as set by `setController`. If no controller has yet been attached, this method returns null

**See Also:**

`setController`

### getKeyframeSequence

`public KeyframeSequence getKeyframeSequence()`

Returns the keyframe sequence object which defines the keyframe values for this animation track.

**Returns:**

the KeyframeSequence object which defines the keyframe values

### getTargetProperty

`public int getTargetProperty()`

Returns the property targeted by this AnimationTrack. The target property is one of the symbolic constants listed above.

**Returns:**

the object property targeted by this track

**javax.microedition.m3g**

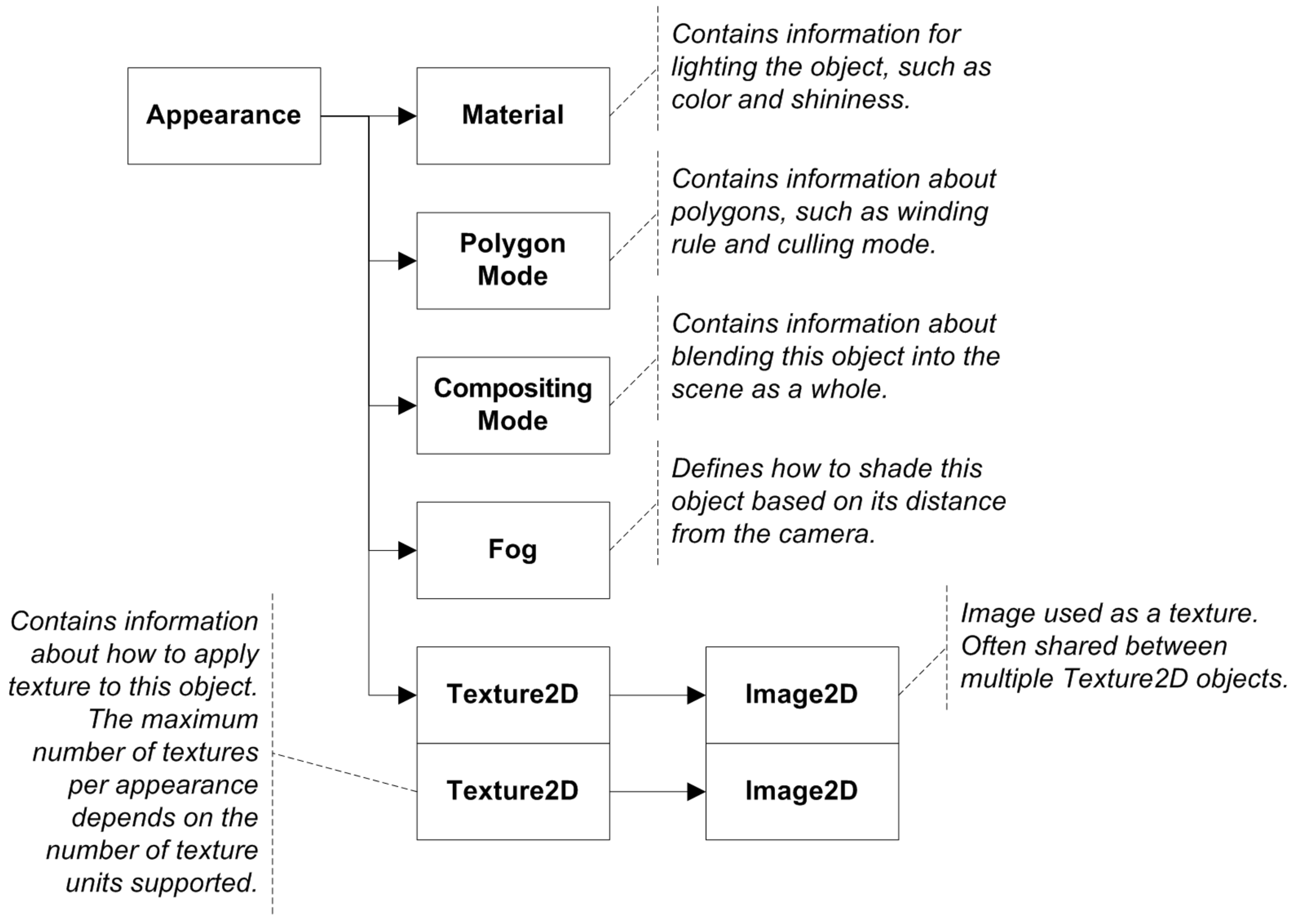
## Class Appearance

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Appearance
```

public class **Appearance**
extends Object3D

A set of component objects that define the rendering attributes of a Mesh or Sprite3D.

Appearance attributes are grouped into component objects, each encapsulating a set of properties that are functionally and logically related to each other. This division helps applications to conserve memory by sharing component objects across multiple meshes and sprites. The Appearance component classes and a summary of their contents are presented in the figure below.


Appearance
Material
Contains information for lighting the object, such as color and shininess.
Polygon Mode
Contains information about polygons, such as winding rule and culling mode.
Compositing Mode
Contains information about blending this object into the scene as a whole.
Fog
Defines how to shade this object based on its distance from the camera.
Contains information about how to apply texture to this object. The maximum number of textures per appearance depends on the number of texture units supported.
Texture2D
Texture2D
Image2D
Image2D
Image used as a texture. Often shared between multiple Texture2D objects.

All components of a newly created Appearance object are initialized to null. It is completely legal for any or all of the components to be null even when rendering. The behavior when each of the components is null is as follows:

- If a Texture2D is null, the corresponding texturing unit is disabled.
- If the PolygonMode is null, default values are used.
- If the CompositingMode is null, default values are used.

- If the Material is null, lighting is disabled.
- If the Fog is null, fogging is disabled.

Using a null Appearance on a submesh (or sprite) disables rendering and picking of that submesh (or sprite). An Appearance must always be provided for an object in order to make it visible.

## Implementation guidelines

By default, everything in Appearance works exactly the same way as in OpenGL 1.3. As a general exception, the color index (palette) mode is not supported. Other than that, any deviations from the OpenGL 1.3 specification are documented explicitly in the individual Appearance component classes.

**See Also:**
Binary format

| Constructor Summary |
|---|
| **`Appearance`**`()`<br>Constructs an Appearance object with default values. |

| Method Summary | |
|---|---|
| `CompositingMode` | **`getCompositingMode`**`()`<br>Returns the current CompositingMode for this Appearance. |
| `Fog` | **`getFog`**`()`<br>Returns the current fogging attributes for this Appearance. |
| `int` | **`getLayer`**`()`<br>Gets the current rendering layer for this Appearance. |
| `Material` | **`getMaterial`**`()`<br>Returns the current Material for this Appearance. |
| `PolygonMode` | **`getPolygonMode`**`()`<br>Returns the current PolygonMode for this Appearance. |
| `Texture2D` | **`getTexture`**`(int index)`<br>Retrieves the current texture image and its attributes for the given texturing unit. |
| `void` | **`setCompositingMode`**`(CompositingMode compositingMode)`<br>Sets the CompositingMode to use for this Appearance. |
| `void` | **`setFog`**`(Fog fog)`<br>Sets the fogging attributes to use for this Appearance. |
| `void` | **`setLayer`**`(int layer)`<br>Sets the rendering layer for this Appearance. |
| `void` | **`setMaterial`**`(Material material)`<br>Sets the Material to use when lighting this Appearance. |
| `void` | **`setPolygonMode`**`(PolygonMode polygonMode)`<br>Sets the PolygonMode to use for this Appearance. |

| | |
|---|---|
| void | `setTexture(int index, Texture2D texture)` Sets the texture image and its attributes for the given texturing unit. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### Appearance

`public Appearance()`

Constructs an Appearance object with default values. The default values are:

- rendering layer : 0
- polygon mode : null (use defaults)
- compositing mode : null (use defaults)
- all textures : null (all texturing units disabled)
- material : null (lighting disabled)
- fog : null (fogging disabled)

## Method Detail

### setLayer

`public void setLayer(int layer)`

Sets the rendering layer for this Appearance. When rendering a World, Group or Mesh, submeshes and sprites are guaranteed to be rendered in the order of ascending layers. That is, *all* submeshes and sprites with an appearance at a lower layer are rendered prior to *any* submeshes or sprites at the higher layers. Furthermore, *all* opaque submeshes and sprites at a specific layer are rendered prior to *any* blended submeshes or sprites at the same layer. A submesh or a sprite is defined to be opaque if it uses the `REPLACE` blending mode (see CompositingMode), and blended otherwise.

Implementations are free to do any additional state sorting as long as the aforementioned constraints are met. To allow implementations to operate as efficiently as possible, applications should only use layering constraints when required. As a result of the rules above, the default layer of zero should be sufficient for most uses. Cases where non-zero layers may be useful include background geometry, sky boxes, lens flares, halos, and other special effects.

Note that the rendering layer has no effect on picking.

**Parameters:**

`layer` - the rendering layer for submeshes and sprites having this Appearance

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `layer` is not in [-63, 63]

**See Also:**

`getLayer`, `Mesh`

### getLayer

`public int getLayer()`

Gets the current rendering layer for this Appearance.

**Returns:**

the current rendering layer; this is always in the range [-63, 63]

**See Also:**

`setLayer`

### setFog

`public void setFog(Fog fog)`

Sets the fogging attributes to use for this Appearance. If the Fog object is set to null, fogging is disabled.

**Parameters:**

`fog` - a Fog object, or null to disable fogging

**See Also:**

`getFog`

### getFog

`public Fog getFog()`

Returns the current fogging attributes for this Appearance.

**Returns:**

the current Fog object, or null if fogging is disabled

**See Also:**

`setFog`

### setPolygonMode

`public void setPolygonMode(PolygonMode polygonMode)`

Sets the PolygonMode to use for this Appearance. If the PolygonMode is set to null, the default values are used.

**Parameters:**

`polygonMode` - a PolygonMode object, or null to use the defaults

**See Also:**

`getPolygonMode`

### getPolygonMode

```
public PolygonMode getPolygonMode()
```

Returns the current PolygonMode for this Appearance.

**Returns:**
the current PolygonMode object, or null if no PolygonMode is set

**See Also:**
`setPolygonMode`

### setCompositingMode

```
public void setCompositingMode(CompositingMode compositingMode)
```

Sets the CompositingMode to use for this Appearance. If the CompositingMode is set to null, the default values are used.

**Parameters:**
`compositingMode` - a CompositingMode object, or null to use the defaults

**See Also:**
`getCompositingMode`

### getCompositingMode

```
public CompositingMode getCompositingMode()
```

Returns the current CompositingMode for this Appearance.

**Returns:**
the current CompositingMode object, or null if no CompositingMode is set

**See Also:**
`setCompositingMode`

### setTexture

```
public void setTexture(int index,
                       Texture2D texture)
```

Sets the texture image and its attributes for the given texturing unit. If the texture object is set to null, the specified texturing unit is disabled.

**Parameters:**
`index` - texturing unit index
`texture` - a texture object for the specified texturing unit, or null to disable the unit

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `index` is not a valid texturing unit index

**See Also:**

`getTexture`

### getTexture

```
public Texture2D getTexture(int index)
```

Retrieves the current texture image and its attributes for the given texturing unit.

**Parameters:**

`index` - texturing unit index

**Returns:**

the current texture object of the specified texturing unit, or null if the unit is disabled

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `index` is not a valid texturing unit index

**See Also:**

`setTexture`

### setMaterial

```
public void setMaterial(Material material)
```

Sets the Material to use when lighting this Appearance. If the Material is set to null, lighting is disabled. See the Material class description for more information.

**Parameters:**

`material` - a Material object, or null to disable lighting

**See Also:**

`getMaterial`

### getMaterial

```
public Material getMaterial()
```

Returns the current Material for this Appearance.

**Returns:**

the current Material object, or null if lighting is disabled

**See Also:**

`setMaterial`

**javax.microedition.m3g**

# Class Background

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Background
```

public class **Background**
extends Object3D

Defines whether and how to clear the viewport.

The portions of the frame buffer that correspond to the current viewport are cleared according to a given Background object. In retained mode (that is, when rendering a World), the Background object associated with the World is used. In immediate mode, a Background object is given as a parameter to `clear`. In absence of a Background object, the default values specified in the constructor are used.

Clearing can be enabled and disabled individually for the color buffer and the depth buffer. The color buffer is cleared using the background color and/or the background image, as specified below. If the background image is set to null (the initial value), only the background color is used. The depth buffer is always cleared to the maximum depth value.

## Background image

The background image is stored as a reference to an Image2D. If the referenced Image2D is modified by the application, or a new Image2D is bound as the background image, the modifications are immediately reflected in the Background object.

The background image must be in RGB or RGBA format. Furthermore, it must be in the same format as the currently bound rendering target. This is enforced by the `render(World)` and `clear` methods in Graphics3D.

A cropping rectangle very similar to that of Sprite3D is available to facilitate scrolling and zooming of the background image. The contents of the crop rectangle are scaled to fill the entire viewport. The crop rectangle need not lie within the source image boundaries. If it does not, the source image is either considered to repeat indefinitely in the image space (the `REPEAT` mode) or to not repeat at all, with pixels outside the source image having the background color (the `BORDER` mode).

Contrary to texture images, the width and height of a background image do not have to be powers of two. Furthermore, the maximum size of a background image is only determined by the amount of available memory; there is no fixed limit. The dimensions of the crop rectangle are also unbounded.

## Implementation guidelines

The requirements and recommendations given in the Implementation guidelines of Sprite3D also apply for Background images. In particular, implementations using textured rectangles to blit the background image must follow the resampling rules specified for sprites.

**See Also:**

Binary format

## Field Summary

| | |
|---|---|
| `static int` | **`BORDER`** Specifies that the imaginary pixels outside of the source image boundaries in X or Y direction are considered to have the background color. |
| `static int` | **`REPEAT`** Specifies that the imaginary pixels outside of the source image boundaries in X or Y direction are considered to have the same color as the pixel in the corresponding position in the source image. |

## Constructor Summary

| |
|---|
| **`Background`**`()` Constructs a new Background with default values. |

## Method Summary

| | |
|---|---|
| `int` | **`getColor`**`()` Retrieves the current background color. |
| `int` | **`getCropHeight`**`()` Gets the current cropping rectangle height within the source image. |
| `int` | **`getCropWidth`**`()` Gets the current cropping rectangle width within the source image. |
| `int` | **`getCropX`**`()` Retrieves the current cropping rectangle X offset relative to the source image top left corner. |
| `int` | **`getCropY`**`()` Retrieves the current cropping rectangle Y offset relative to the source image top left corner. |
| `Image2D` | **`getImage`**`()` Gets the current background image. |
| `int` | **`getImageModeX`**`()` Gets the current background image repeat mode for the X dimension. |
| `int` | **`getImageModeY`**`()` Gets the current background image repeat mode for the Y dimension. |
| `boolean` | **`isColorClearEnabled`**`()` Queries whether color buffer clearing is enabled. |
| `boolean` | **`isDepthClearEnabled`**`()` Queries whether depth buffer clearing is enabled. |
| `void` | **`setColor`**`(int ARGB)` Sets the background color. |
| `void` | **`setColorClearEnable`**`(boolean enable)` Enables or disables color buffer clearing. |
| `void` | **`setCrop`**`(int cropX, int cropY, int width, int height)` Sets a cropping rectangle within the background image. |

| | |
|---|---|
| void | **setDepthClearEnable**(boolean enable)<br>Enables or disables depth buffer clearing. |
| void | **setImage**(Image2D image)<br>Sets the background image, or switches from background image mode to background color mode. |
| void | **setImageMode**(int modeX, int modeY)<br>Sets the background image repeat mode for the X and Y directions. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

### BORDER

`public static final int BORDER`

Specifies that the imaginary pixels outside of the source image boundaries in X or Y direction are considered to have the background color.

**See Also:**
Constant Field Values

### REPEAT

`public static final int REPEAT`

Specifies that the imaginary pixels outside of the source image boundaries in X or Y direction are considered to have the same color as the pixel in the corresponding position in the source image. Formally, a pixel at position X will have the same color as the pixel at position X % N, where N is the width or height of the image and % is the modulo operator.

**See Also:**
Constant Field Values

## Constructor Detail

### Background

`public Background()`

Constructs a new Background with default values. The default values are:

- color clear enable : *true* (clear the color buffer)
- depth clear enable : *true* (clear the depth buffer)
- background color : 0x00000000 (black, transparent)
- background image : null (use the background color only)
- background image mode : `BORDER, BORDER`
- crop rectangle : undefined (reset at `setImage`

## Method Detail

### setColorClearEnable

```
public void setColorClearEnable(boolean enable)
```

Enables or disables color buffer clearing. If color buffer clearing is enabled, the portion of the color buffer that corresponds to the viewport is cleared with the background image and/or the background color.

**Parameters:**
`enable` - *true* to enable color buffer clearing; *false* to disable

### isColorClearEnabled

```
public boolean isColorClearEnabled()
```

Queries whether color buffer clearing is enabled.

**Returns:**
*true* if color buffer clearing is enabled; *false* if it is disabled

### setDepthClearEnable

```
public void setDepthClearEnable(boolean enable)
```

Enables or disables depth buffer clearing. If depth buffer clearing is enabled, the portion of the depth buffer that corresponds to the viewport is cleared to the maximum depth value.

This setting is ignored if depth buffering is disabled in Graphics3D (see `bindTarget`).

**Parameters:**
`enable` - *true* to enable depth buffer clearing; *false* to disable

### isDepthClearEnabled

```
public boolean isDepthClearEnabled()
```

Queries whether depth buffer clearing is enabled.

**Returns:**

*true* if depth buffer clearing is enabled; *false* if it is disabled

## setColor

```
public void setColor(int ARGB)
```

Sets the background color. This is the color that the imaginary pixels outside of the source image boundaries are considered to have in the `BORDER` mode. If there is no background image, the viewport is cleared with this color only.

The alpha component of the background color is ignored when rendering to an RGB target.

**Parameters:**

`ARGB` - the new background color in 0xAARRGGBB format

**See Also:**

`getColor`

## getColor

```
public int getColor()
```

Retrieves the current background color.

**Returns:**

the current background color in 0xAARRGGBB format

**See Also:**

`setColor`

## setImage

```
public void setImage(Image2D image)
```

Sets the background image, or switches from background image mode to background color mode. The background image must be in the same format as the rendering target: RGB or RGBA in case of an Image2D target and RGB in case of a MIDP Graphics target.

The crop rectangle is set such that its top left corner is at the top left corner of the image, and its width and height are equal to the dimensions of the image.

**Parameters:**

`image` - the background image, or null to disable the current background image (if any) and clear with the background color only

**Throws:**

`java.lang.IllegalArgumentException` - if `image` is not in RGB or RGBA format

**See Also:**

`getImage`

## getImage

```
public Image2D getImage()
```

Gets the current background image.

**Returns:**
the current background image

**See Also:**
`setImage`

## setImageMode

```
public void setImageMode(int modeX,
                         int modeY)
```

Sets the background image repeat mode for the X and Y directions.

**Parameters:**
`modeX` - X repeat mode; one of `BORDER, REPEAT`
`modeY` - Y repeat mode; one of `BORDER, REPEAT`

**Throws:**
`java.lang.IllegalArgumentException` - if `modeX` or `modeY` is not one of the enumerated values listed above

## getImageModeX

```
public int getImageModeX()
```

Gets the current background image repeat mode for the X dimension.

**Returns:**
the X repeat mode

**See Also:**
`setImageMode`

## getImageModeY

```
public int getImageModeY()
```

Gets the current background image repeat mode for the Y dimension.

**Returns:**
the Y repeat mode

**See Also:**
`setImageMode`

## setCrop

```
public void setCrop(int cropX,
                    int cropY,
                    int width,
                    int height)
```

Sets a cropping rectangle within the background image. The contents of the crop rectangle are scaled (stretched or condensed) to fill the viewport entirely.

The position of the upper left corner of the crop rectangle is given in pixels, relative to the upper left corner of the Image2D. The relative position may be negative in either or both axes. The width and height of the crop rectangle are also given in pixels, and must not be negative. If either of them is zero, the color buffer is cleared with the background color only.

If the crop rectangle lies completely or partially outside of the source image boundaries, the values of the (imaginary) pixels outside of the image are defined by the repeat mode. In `BORDER` mode, the imaginary pixels are taken to have the background color. In `REPEAT` mode, the source image is considered to repeat indefinitely. The repeat mode can be specified independently for the X and Y directions.

**Parameters:**
`cropX` - the X position of the top left of the crop rectangle, in pixels
`cropY` - the Y position of the top left of the crop rectangle, in pixels
`width` - the width of the crop rectangle, in pixels
`height` - the height of the crop rectangle, in pixels
**Throws:**
`java.lang.IllegalArgumentException` - if `width < 0`
`java.lang.IllegalArgumentException` - if `height < 0`

## getCropX

`public int **getCropX**()`

Retrieves the current cropping rectangle X offset relative to the source image top left corner.

**Returns:**
the X offset of the cropping rectangle
**See Also:**
`setCrop`

## getCropY

`public int **getCropY**()`

Retrieves the current cropping rectangle Y offset relative to the source image top left corner.

**Returns:**
the X offset of the cropping rectangle
**See Also:**
`setCrop`

## getCropWidth

```
public int getCropWidth()
```

Gets the current cropping rectangle width within the source image.

**Returns:**
the width of the cropping rectangle

**See Also:**
`setCrop`

## getCropHeight

```
public int getCropHeight()
```

Gets the current cropping rectangle height within the source image.

**Returns:**
the height of the cropping rectangle

**See Also:**
`setCrop`

**javax.microedition.m3g**

## Class Camera

```
java.lang.Object
  └─javax.microedition.m3g.Object3D
      └─javax.microedition.m3g.Transformable
          └─javax.microedition.m3g.Node
              └─javax.microedition.m3g.Camera
```

public class **Camera**
extends Node

A scene graph node that defines the position of the viewer in the scene and the projection from 3D to 2D.

The camera is always facing towards the negative Z axis, (0 0 -1), in its local coordinate system. The camera can be positioned and oriented in the same way as any other Node; that is, using the node transformations of the camera node and its ancestors.

The projection matrix transforms homogeneous (4D) coordinates from *camera space* to *clip space*. Triangles are then clipped to the view volume, which is defined by

$$-w \le x \le w$$
$$-w \le y \le w$$
$$-w \le z \le w$$

where (x y z w) are the clip-space coordinates of each vertex. A polygon is discarded by the clipper if all of its vertices have a negative W value. If a polygon crosses the W = 0 boundary, the portion of the polygon that lies on the negative side is discarded.

Subsequent to clipping, X, Y, and Z are divided by W to obtain *normalized device coordinates* (NDC). These are between [-1, 1], and the center of the viewport lies at the origin. Finally, the viewport mapping and the depth range are applied to transform the normalized X, Y and Z into *window coordinates*. The viewport and depth range mappings are specified in Graphics3D.

### Implementation guidelines

Clipping is done according to the OpenGL 1.3 specification, section 2.11, with the exception that user-defined clip planes are not supported. Clipping of colors and texture coordinates is done according to section 2.13.8.

To clarify the handling of polygons with negative clip-space W, we deviate slightly from the OpenGL specification by not only allowing implementations to discard any and all portions of polygons that lie in the region W < 0, but actually requiring them to do so.

**See Also:**
Binary format

## Field Summary

| | |
|---|---|
| static int | **GENERIC**<br>Specifies a generic 4x4 projection matrix. |
| static int | **PARALLEL**<br>Specifies a parallel projection matrix. |
| static int | **PERSPECTIVE**<br>Specifies a perspective projection matrix. |

**Fields inherited from class javax.microedition.m3g.Node**

`NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS`

## Constructor Summary

**Camera**()
Constructs a new Camera node with default values.

## Method Summary

| | |
|---|---|
| int | **getProjection**(float[] params)<br>Gets the current projection parameters and type. |
| int | **getProjection**(Transform transform)<br>Gets the current projection matrix and type. |
| void | **setGeneric**(Transform transform)<br>Sets the given 4x4 transformation as the current projection matrix. |
| void | **setParallel**(float fovy, float aspectRatio, float near, float far)<br>Constructs a parallel projection matrix and sets that as the current projection matrix. |
| void | **setPerspective**(float fovy, float aspectRatio, float near, float far)<br>Constructs a perspective projection matrix and sets that as the current projection matrix. |

**Methods inherited from class javax.microedition.m3g.Node**

`align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope`

**Methods inherited from class javax.microedition.m3g.Transformable**

`getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate`

**Methods inherited from class javax.microedition.m3g.Object3D**

`addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject`

## Field Detail

### GENERIC

`public static final int` **`GENERIC`**

Specifies a generic 4x4 projection matrix.

**See Also:**
Constant Field Values

### PARALLEL

`public static final int` **`PARALLEL`**

Specifies a parallel projection matrix.

**See Also:**
Constant Field Values

### PERSPECTIVE

`public static final int` **`PERSPECTIVE`**

Specifies a perspective projection matrix.

**See Also:**
Constant Field Values

## Constructor Detail

### Camera

`public` **`Camera`**`()`

Constructs a new Camera node with default values. The default values are as follows:

- projection mode : `GENERIC`
- projection matrix : identity

## Method Detail

### setParallel

```
public void setParallel(float fovy,
                        float aspectRatio,
                        float near,
                        float far)
```

Constructs a parallel projection matrix and sets that as the current projection matrix. Note that the near and far clipping planes may be in arbitrary order, although usually near < far.

Denoting the width, height and depth of the view volume by *w*, *h* and *d*, respectively, the parallel projection matrix **P** is constructed as follows.

```
2/w      0         0          0
 0      2/h        0          0
 0       0       -2/d   -(near+far)/d
 0       0         0          1
```

where

h = height (= fovy)
w = aspectRatio * h
d = far - near

The rendered image will "stretch" to fill the viewport entirely (not just the visible portion of it). It is therefore recommended that the aspect ratio given here be equal to the aspect ratio of the viewport as defined in `setViewport`. Otherwise, the image will appear elongated in either the horizontal or the vertical direction. No attempt is made to correct this effect automatically, for example by adjusting the field of view. Instead, the adjustment is left for the application developer to handle as he or she prefers.

In the special case when the near and far distance are equal, the view volume has, in fact, no volume and nothing is rendered. Implementations must detect this rather than trying to construct the projection matrix, as that would result in a divide by zero error.

**Parameters:**
`fovy` - height of the view volume in camera coordinates
`aspectRatio` - aspect ratio of the viewport, that is, width divided by height
`near` - distance to the front clipping plane in camera space
`far` - distance to the back clipping plane in camera space
**Throws:**
`java.lang.IllegalArgumentException` - if `height <= 0`
`java.lang.IllegalArgumentException` - if `aspectRatio <= 0`

### setPerspective

```
public void setPerspective(float fovy,
                           float aspectRatio,
                              float near,
                              float far)
```

Constructs a perspective projection matrix and sets that as the current projection matrix. Note that the near and far clipping planes may be in arbitrary order, although usually near < far. If near and far are equal, nothing is rendered.

The perspective projection matrix **P** is constructed as follows.

```
1/w         0           0              0
 0         1/h          0              0
 0          0      -(near+far)/d   -2*near*far/d
 0          0          -1              0
```

where

h = tan(fovy/2)
w = aspectRatio * h
d = far - near

The rendered image will "stretch" to fill the viewport entirely (not just the visible portion of it). It is therefore recommended that the aspect ratio given here be equal to the aspect ratio of the viewport as defined in `setViewport`. Otherwise, the image will appear elongated in either the horizontal or the vertical direction. No attempt is made to correct this effect automatically, for example by adjusting the field of view. Instead, the adjustment is left for the application developer to handle as he or she prefers.

In the special case when the near and far distance are equal, the view volume has, in fact, no volume and nothing is rendered. Implementations must detect this rather than trying to construct the projection matrix, as that would result in a divide by zero error.

**Parameters:**
`fovy` - field of view in the vertical direction, in degrees
`aspectRatio` - aspect ratio of the viewport, that is, width divided by height
`near` - distance to the front clipping plane
`far` - distance to the back clipping plane

**Throws:**
`java.lang.IllegalArgumentException` - if any argument is `<= 0`
`java.lang.IllegalArgumentException` - if `fovy >= 180`

## setGeneric

```
public void setGeneric(Transform transform)
```

Sets the given 4x4 transformation as the current projection matrix. The contents of the given transformation are copied in, so any further changes to it will not affect the projection matrix.

Generic 4x4 projection matrices are needed for various rendering tricks and speed-up techniques that otherwise could not be implemented at all, or not without incurring significant processing overhead. These include, for example, viewing an arbitrarily large scene by setting the far clipping plane to infinity; rendering a large image in pieces using oblique projection; portals; TV screens and other re-projection cases; stereoscopic rendering; and some shadow algorithms.

**Parameters:**

`transform` - a Transform object to copy as the new projection matrix

**Throws:**

`java.lang.NullPointerException` - if `transform` is null

## getProjection

```
public int getProjection(Transform transform)
```

Gets the current projection matrix and type. This method is available regardless of the type of projection, since parallel and perspective projections can always be returned in the 4x4 matrix form.

**Parameters:**

`transform` - a Transform object to populate with the matrix, or null to only return the type of projection

**Returns:**

the type of projection: `GENERIC`, `PERSPECTIVE`, or `PARALLEL`

**Throws:**

`java.lang.ArithmeticException` - if the transformation matrix cannot be computed due to illegal perspective or parallel projection parameters (that is, if `near == far`)

## getProjection

```
public int getProjection(float[] params)
```

Gets the current projection parameters and type. The given float array is populated with the projection parameters in the same order as they are supplied to the respective set methods, `setPerspective` and `setParallel`. If the projection type is `GENERIC`, the float array is left untouched. This is the case even if the generic projection matrix actually is a perspective or parallel projection.

**Parameters:**

`params` - float array to fill in with the four projection parameters, or null to only return the type of projection

**Returns:**

the type of projection: `GENERIC`, `PERSPECTIVE`, or `PARALLEL`

**Throws:**

`java.lang.IllegalArgumentException` - if `(params != null) && (params.length < 4)`

**javax.microedition.m3g**

## Class CompositingMode

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.CompositingMode
```

public class **CompositingMode**
extends Object3D

An Appearance component encapsulating per-pixel compositing attributes.

*Depth offset* is added to the depth (Z) value of a pixel prior to depth test and depth write. The offset is constant across a polygon. Depth offset is used to prevent *Z fighting*, which makes coplanar polygons intersect each other on the screen due to the limited resolution of the depth buffer. Depth offset allows, for example, white lines on a highway or scorch marks on a wall (*decals* in general) to be implemented with polygons instead of textures. Depth offset has no effect if depth buffering is disabled.

*Blending* combines the incoming fragment's R, G, B, and A values with the R, G, B, and A values stored in the frame buffer at the incoming fragment's location. The table below defines the available blending modes, in terms of the source color $C_s = (R_s, G_s, B_s, A_s)$ and the destination color $C_d = (R_d, G_d, B_d, A_d)$. The source color is the incoming fragment's color value, while the destination color is the pre-existing color value in the frame buffer. The corresponding OpenGL source and destination blend functions are included in the table for reference.

| Mode | Definition | OpenGL src blend func | OpenGL dst blend func |
|---|---|---|---|
| `REPLACE` | $C_d = C_s$ | `ONE` | `ZERO` |
| `ALPHA_ADD` | $C_d = C_s A_s + C_d$ | `SRC_ALPHA` | `ONE` |
| `ALPHA` | $C_d = C_s A_s + C_d (1 - A_s)$ | `SRC_ALPHA` | `ONE_MINUS_SRC_ALPHA` |
| `MODULATE` | $C_d = C_s C_d$ | `DST_COLOR` | `ZERO` |
| `MODULATE_X2` | $C_d = 2 C_s C_d$ | `DST_COLOR` | `SRC_COLOR` |

### Implementation guidelines

Depth offset is computed according to section 3.5.5 in the OpenGL 1.3 specification. Per-fragment operations are done according to sections 4.1 and 4.2, with the following exceptions:

- The alpha test function is always `GEQUAL`;
- Stencil testing and the stencil buffer are not supported;
- The depth test function is always `LEQUAL`;
- The blend equation is always `FUNC_ADD`;
- Blend function combinations are limited to the ones listed above;
- The constant blend color is not supported;
- Logical operations are not supported;
- Individual masking of R, G and B is not supported;
- The accumulation buffer is not supported.

Multisampling is not supported explicitly, but implementations may use it internally to implement full-scene antialiasing. The full-scene antialiasing hint can be enabled or disabled in Graphics3D. Similarly, the dithering hint in Graphics3D may be implemented using the per-fragment dithering feature of OpenGL.

**See Also:**
Binary format

## Field Summary

| | |
|---|---|
| static int | **ALPHA**<br>Selects the alpha blend mode. |
| static int | **ALPHA_ADD**<br>Selects the additive blend mode. |
| static int | **MODULATE**<br>Selects the basic modulating blending mode; source pixels are multiplied with the destination pixels. |
| static int | **MODULATE_X2**<br>Selects the brighter modulating blending mode. |
| static int | **REPLACE**<br>Selects the replace mode. |

## Constructor Summary

| |
|---|
| **CompositingMode**()<br>Constructs a CompositingMode object with default values. |

## Method Summary

| | |
|---|---|
| float | **getAlphaThreshold**()<br>Retrieves the current alpha testing threshold. |
| int | **getBlending**()<br>Retrieves the current frame buffer blending mode. |
| float | **getDepthOffsetFactor**()<br>Retrieves the current depth offset slope factor. |
| float | **getDepthOffsetUnits**()<br>Retrieves the current constant depth offset. |
| boolean | **isAlphaWriteEnabled**()<br>Queries whether alpha writing is enabled. |
| boolean | **isColorWriteEnabled**()<br>Queries whether color writing is enabled. |
| boolean | **isDepthTestEnabled**()<br>Queries whether depth testing is enabled. |
| boolean | **isDepthWriteEnabled**()<br>Queries whether depth writing is enabled. |

| | |
|---|---|
| void | `setAlphaThreshold(float threshold)` Sets the threshold value for alpha testing. |
| void | `setAlphaWriteEnable(boolean enable)` Enables or disables writing of fragment alpha values into the color buffer. |
| void | `setBlending(int mode)` Selects a method of combining the pixel to be rendered with the pixel already in the frame buffer. |
| void | `setColorWriteEnable(boolean enable)` Enables or disables writing of fragment color values into the color buffer. |
| void | `setDepthOffset(float factor, float units)` Defines a value that is added to the screen space Z coordinate of a pixel immediately before depth test and depth write. |
| void | `setDepthTestEnable(boolean enable)` Enables or disables depth testing. |
| void | `setDepthWriteEnable(boolean enable)` Enables or disables writing of fragment depth values into the depth buffer. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

### ALPHA

`public static final int` **`ALPHA`**

Selects the alpha blend mode. A weighted average of the source and destination pixels is computed.

**See Also:**
Constant Field Values

### ALPHA_ADD

`public static final int` **`ALPHA_ADD`**

Selects the additive blend mode. The source pixel is first scaled by the source alpha and then summed with the destination pixel.

**See Also:**
Constant Field Values

### MODULATE

```
public static final int MODULATE
```

Selects the basic modulating blending mode; source pixels are multiplied with the destination pixels.

**See Also:**
Constant Field Values

### MODULATE_X2

```
public static final int MODULATE_X2
```

Selects the brighter modulating blending mode. This is the same as basic modulation, but the results are multiplied by two (and saturated to 1.0) to compensate for the loss of luminance caused by the component-wise multiplication.

**See Also:**
Constant Field Values

### REPLACE

```
public static final int REPLACE
```

Selects the replace mode. The destination pixel is overwritten with the source pixel.

**See Also:**
Constant Field Values

## Constructor Detail

### CompositingMode

```
public CompositingMode()
```

Constructs a CompositingMode object with default values. The default values are:

- blending mode : `REPLACE`
- alpha threshold : 0.0
- depth offset : 0.0, 0.0
- depth test : enabled
- depth write : enabled
- color write : enabled
- alpha write : enabled

## Method Detail

## setBlending

`public void **setBlending**(int mode)`

Selects a method of combining the pixel to be rendered with the pixel already in the frame buffer. Blending is applied as the very last step of the pixel processing pipeline.

**Parameters:**
`mode` - the new blending mode
**Throws:**
`java.lang.IllegalArgumentException` - if `mode` is not one of the symbolic constants listed above
**See Also:**
`getBlending`

## getBlending

`public int **getBlending**()`

Retrieves the current frame buffer blending mode.

**Returns:**
the currently active blending mode
**See Also:**
`setBlending`

## setAlphaThreshold

`public void **setAlphaThreshold**(float threshold)`

Sets the threshold value for alpha testing. If the alpha component of a fragment is less than the alpha threshold, the fragment is not rendered. Consequently, if the threshold is 1.0, only fragments with the maximum alpha value (1.0) will be drawn, and if the threshold is 0.0, all fragments will be drawn.

**Parameters:**
`threshold` - the new alpha threshold; must be [0, 1]
**Throws:**
`java.lang.IllegalArgumentException` - if `threshold` is negative or greater than 1.0
**See Also:**
`getAlphaThreshold`

## getAlphaThreshold

`public float **getAlphaThreshold**()`

Retrieves the current alpha testing threshold.

**Returns:**

the current alpha threshold [0, 1]

**See Also:**

`setAlphaThreshold`

### setAlphaWriteEnable

`public void setAlphaWriteEnable(boolean enable)`

Enables or disables writing of fragment alpha values into the color buffer.

**Parameters:**

`enable` - *true* to enable alpha write; *false* to disable it

**See Also:**

`setColorWriteEnable`

### isAlphaWriteEnabled

`public boolean isAlphaWriteEnabled()`

Queries whether alpha writing is enabled.

**Returns:**

*true* if alpha writing is enabled; *false* if it's disabled

**See Also:**

`isColorWriteEnabled`

### setColorWriteEnable

`public void setColorWriteEnable(boolean enable)`

Enables or disables writing of fragment color values into the color buffer.

**Parameters:**

`enable` - *true* to enable color write; *false* to disable it

**See Also:**

`setAlphaWriteEnable`

### isColorWriteEnabled

`public boolean isColorWriteEnabled()`

Queries whether color writing is enabled.

**Returns:**

*true* if color writing is enabled; *false* if it's disabled

**See Also:**

`isAlphaWriteEnabled`

## setDepthWriteEnable

```
public void setDepthWriteEnable(boolean enable)
```

Enables or disables writing of fragment depth values into the depth buffer. If depth buffering is not enabled in the current Graphics3D, this setting has no effect; nothing will be written anyway.

If both depth testing and depth writing are enabled, and a fragment passes the depth test, that fragment's depth value is written to the depth buffer at the corresponding position.

If depth testing is disabled and depth writing is enabled, a fragment's depth value is always written to the depth buffer.

If depth writing is disabled, a fragment's depth value is never written to the depth buffer.

**Parameters:**

`enable` - *true* to enable depth write; *false* to disable it

## isDepthWriteEnabled

```
public boolean isDepthWriteEnabled()
```

Queries whether depth writing is enabled.

**Returns:**

*true* if depth writing is enabled; *false* if it's disabled

## setDepthTestEnable

```
public void setDepthTestEnable(boolean enable)
```

Enables or disables depth testing. If depth testing is enabled, a fragment is written to the frame buffer if and only if its depth component is less than or equal to the corresponding value in the depth buffer. If there is no depth buffer in the current rendering target, this setting has no effect; the fragment will be written anyway.

**Parameters:**

`enable` - *true* to enable depth test; *false* to disable it

## isDepthTestEnabled

```
public boolean isDepthTestEnabled()
```

Queries whether depth testing is enabled.

**Returns:**

*true* if depth testing is enabled; *false* if it's disabled

## setDepthOffset

```
public void setDepthOffset(float factor,
                           float units)
```

Defines a value that is added to the screen space Z coordinate of a pixel immediately before depth test and depth write. The depth offset is computed for each polygon with the following formula:

offset = m * factor + r * units
r = smallest distinguishable depth buffer increment
m = maximum depth slope (Z gradient) of the triangle

**Parameters:**
`factor` - slope dependent depth offset
`units` - constant depth offset

**See Also:**
`getDepthOffsetFactor`, `getDepthOffsetUnits`

### getDepthOffsetFactor

```
public float getDepthOffsetFactor()
```

Retrieves the current depth offset slope factor. This is the `factor` parameter set in `setDepthOffset`.

**Returns:**
the current depth offset factor

**See Also:**
`setDepthOffset`

### getDepthOffsetUnits

```
public float getDepthOffsetUnits()
```

Retrieves the current constant depth offset. This is the `units` parameter set in `setDepthOffset`.

**Returns:**
the current depth offset in depth units

**See Also:**
`setDepthOffset`

**javax.microedition.m3g**

## Class Fog

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Fog
```

public class **Fog**
extends Object3D

An Appearance component encapsulating attributes for fogging.

### Implementation guidelines

Fogging is done according to the OpenGL 1.3 specification, section 3.10, with the exception that the EXP2 mode is not supported. The same approximations in fog computation are allowed as in OpenGL: Firstly, the fog function may be evaluated at vertices and then interpolated to obtain the per-fragment values, and secondly, the distance from the camera to the fragment center may be approximated with the fragment's Z coordinate.

**See Also:**
Binary format

| **Field Summary** | |
|---|---|
| `static int` | **`EXPONENTIAL`**<br>A parameter to `setMode`, specifying exponential fog. |
| `static int` | **`LINEAR`**<br>A parameter to `setMode`, specifying linear fog. |

| **Constructor Summary** |
|---|
| **`Fog`**`()`<br>Constructs a new Fog object with default values. |

| **Method Summary** | |
|---|---|
| `int` | **`getColor`**`()`<br>Retrieves the current color of this Fog. |
| `float` | **`getDensity`**`()`<br>Retrieves the fog density of exponential fog. |
| `float` | **`getFarDistance`**`()`<br>Retrieves the linear fog far distance. |
| `int` | **`getMode`**`()`<br>Retrieves the current fog mode. |

| | |
|---|---|
| `float` | **`getNearDistance`**`()`<br>Retrieves the linear fog near distance. |
| `void` | **`setColor`**`(int RGB)`<br>Sets the color of this Fog. |
| `void` | **`setDensity`**`(float density)`<br>Sets the fog density for exponential fog. |
| `void` | **`setLinear`**`(float near, float far)`<br>Sets the near and far distances for linear fog. |
| `void` | **`setMode`**`(int mode)`<br>Sets the fog mode to either linear or exponential. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

### EXPONENTIAL

`public static final int` **`EXPONENTIAL`**

A parameter to `setMode`, specifying exponential fog. The fog blending factor *f* is calculated according to the formula:

$$f = e^{-dz}$$

where z is the distance, in camera coordinates, from the camera origin to the fragment center, and d is the fog density set in `setDensity`. The result is clamped to the [0, 1] range.

**See Also:**
Constant Field Values

### LINEAR

`public static final int` **`LINEAR`**

A parameter to `setMode`, specifying linear fog. The fog blending factor *f* is calculated according to the formula:

$$f = (far - z) / (far - near)$$

where z is the distance, in camera coordinates, from the camera origin to the fragment center, and near and far are the distances set in `setLinear`. The result is clamped to the [0, 1] range. If far == near, that is, the far and near distances are equal, the result is undefined.

The smaller the fog blending factor is, the more of the fog color is blended in to the rasterized fragment. The blending factor reaches its minimum at the far plane, and the maximum at the near plane.

**See Also:**
Constant Field Values

## Constructor Detail

### Fog

`public **Fog**()`

Constructs a new Fog object with default values. The default values are:

- mode : `LINEAR`
- density : 1.0 (exponential fog only)
- near distance : 0.0 (linear fog only)
- far distance : 1.0 (linear fog only)
- color: 0x00000000

## Method Detail

### setMode

`public void **setMode**(int mode)`

Sets the fog mode to either linear or exponential.

**Parameters:**
`mode` - the fog mode to set; one of the symbolic constants listed above
**Throws:**
`java.lang.IllegalArgumentException` - if `mode` is not `LINEAR` or `EXPONENTIAL`
**See Also:**
`getMode`

### getMode

`public int **getMode**()`

Retrieves the current fog mode.

**Returns:**
the current fog mode; one of the symbolic constants listed above
**See Also:**
`setMode`

## setLinear

```
public void setLinear(float near,
                      float far)
```

Sets the near and far distances for linear fog. Note that the near distance does not have to be smaller than the far distance, although that is usually the case.

Note that this setting has no effect on rendering unless the type of this Fog is (or is later set to) `LINEAR`.

**Parameters:**
`near` - distance to the linear fog near plane
`far` - distance to the linear fog far plane

## getNearDistance

```
public float getNearDistance()
```

Retrieves the linear fog near distance.

**Returns:**
the current distance to the linear fog near plane
**See Also:**
`setLinear`

## getFarDistance

```
public float getFarDistance()
```

Retrieves the linear fog far distance.

**Returns:**
the current distance to the linear fog far plane
**See Also:**
`setLinear`

## setDensity

```
public void setDensity(float density)
```

Sets the fog density for exponential fog.

Note that this setting has no effect on rendering unless the type of this Fog is (or is later set to) `EXPONENTIAL`.

**Parameters:**
`density` - the density to set for this Fog
**Throws:**

`java.lang.IllegalArgumentException` - if `density < 0`

**See Also:**

`getDensity`

## getDensity

```
public float getDensity()
```

Retrieves the fog density of exponential fog.

**Returns:**

the current density of this Fog

**See Also:**

`setDensity`

## setColor

```
public void setColor(int RGB)
```

Sets the color of this Fog. The high order byte of the color value (that is, the alpha component) is ignored.

**Parameters:**

`RGB` - the color to set for this Fog in 0x00RRGGBB format

**See Also:**

`getColor`

## getColor

```
public int getColor()
```

Retrieves the current color of this Fog. The high order byte of the color value (that is, the alpha component) is guaranteed to be zero.

**Returns:**

the current color of this Fog in 0x00RRGGBB format

**See Also:**

`setColor`

**javax.microedition.m3g**

# Class Graphics3D

```
java.lang.Object
  └─ javax.microedition.m3g.Graphics3D
```

public class **Graphics3D**
extends java.lang.Object

A singleton 3D graphics context that can be bound to a rendering target. All rendering is done through the `render` methods in this class, including the rendering of World objects. There is no other way to draw anything in this API.

## Quick introduction

Using the Graphics3D is very straightforward. The application only needs to obtain the Graphics3D instance (there is only one), bind a target to it, render everything, and release the target. This is shown in the code fragment below.

```
public class MyCanvas extends Canvas
{
   Graphics3D myG3D = Graphics3D.getInstance();

   public void paint(Graphics g) {
   try {
      myG3D.bindTarget(g);
      // ... update the scene ...
      // ... render the scene ...
   } finally {
      myG3D.releaseTarget();
   }
}
```

## Immediate mode and retained mode rendering

There are four different `render` methods, operating at different levels of granularity. The first method is for rendering an entire World. When this method is used, we say that the API operates in *retained mode*. The second method is for rendering scene graph nodes, including Groups. The third and fourth methods are for rendering an individual submesh. When the node and submesh rendering methods are used, the API is said to operate in *immediate mode*.

There is a *current camera* and an array of *current lights* in Graphics3D. These are used by the immediate mode rendering methods only. The retained mode rendering method `render(World)` uses the camera and lights that are specified in the World itself, ignoring the Graphics3D camera and lights. Instead, `render(World)` *replaces* the Graphics3D current camera and lights with the active camera and lights in the rendered World. This allows subsequent immediate mode rendering to utilize the same camera and lighting setup as the World.

## Rendering targets

Before rendering anything or even clearing the screen, the application must bind a *rendering target* to this Graphics3D, using the `bindTarget` method. When finished with rendering a frame, the application must release the rendering target by calling the `releaseTarget` method. Implementations may queue rendering commands and only execute them when the target is released.

The rendering target can be either a Graphics object or an Image2D. The type of the Graphics object is specific to the Java profile that this API is implemented on. In case of the MID profile, it must be a `javax.microedition.lcdui.Graphics` object, and it may be associated with a Canvas, mutable Image, or CustomItem.

Once a rendering target is bound to the Graphics3D, all rendering will end up in the color buffer of its rendering target until `releaseTarget` is called. If the `OVERWRITE` hint flag is not given, the contents of the rendering target, after releasing it, will be equal to what they were before the target was bound, augmented with any 3D rendering performed while it was bound. If the hint flag is given, the implementation may substitute undefined data for the original contents of the rendering target.

There can be only one rendering target bound to the Graphics3D at a time. Also, a bound rendering target should not be accessed via any other interface than the host Graphics3D. This is not enforced, but the results are unpredictable otherwise. For example, the following scenarios will result in unpredictable output:

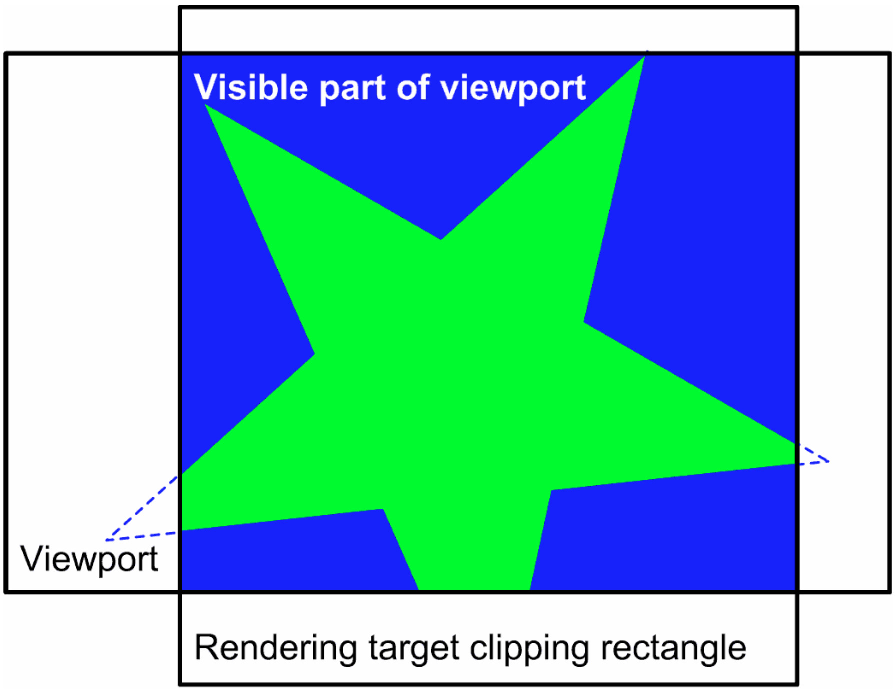
- 2D graphics is rendered via MIDP into a bound Image or Canvas.
- A bound Image is read from by the application or the MIDP implementation.
- A bound Image2D is used by a Graphics3D `render` method.

The contents of the depth buffer are unspecified after `bindTarget`, and they are discarded after `releaseTarget`. In order to clear depth buffer contents (and color buffer contents, if so desired) after binding a rendering target, the application must call the `clear` method, either explicitly, or implicitly by rendering a World.

## Origin translation and clipping

The viewport can be freely positioned relative to the rendering target, without releasing and re-binding the target. The position of the viewport is specified relative to the origin of the rendering target. For Graphics targets, this is the origin in effect when calling `bindTarget`; for Image2D targets, the origin is always at the top left corner. Changing the origin of a bound Graphics object has no effect.

All 3D rendering is clipped to visible part of the viewport, that is, the intersection of the viewport specified in `setViewport` and the rendering target clip rectangle. Rendering operations (including `clear`) must not touch pixels falling outside of the visible part of the viewport. This is illustrated in the figure below.



For Graphics targets, the clipping rectangle is the MIDP/AWT clipping rectangle that is in effect when calling `bindTarget`. Similar to the origin, changing the clipping rectangle of a bound Graphics object will result in unpredictable behavior. For Image2D targets, the clipping rectangle comprises all pixels in the target image.

Origin translation and clipping are independent of the viewport and projection transformations, as well as rasterization. All other parameters being equal, rendering calls must produce the same pixels (prior to clipping) into the area bounded by the viewport regardless of the position of the viewport or the target clipping rectangle.

Note that when we refer to the viewport in this specification, we occasionally mean only the visible part of it. If it is not obvious from the context whether we mean the full viewport or just the visible portion, we state that explicitly.

## Rendering quality hints

In some situations, image quality might be more important for an application than rendering speed or memory usage. Some applications might also want to increase or decrease the image quality based on device capabilities. Some might go so far as to dynamically adjust the quality; for instance, by displaying freeze frames or slow-motion sequences in higher quality.

There are three global options in Graphics3D that allow applications to indicate a preference for higher rendering quality at the expense of slower speed and/or extra memory usage. The application can specify these *rendering quality hints* when binding a rendering target (see `bindTarget`), and query their availability using `getProperties`. The hints are as follows:

- **Antialiasing**. Specifies that antialiasing should be used to increase the perceived resolution of the screen. No particular method of antialiasing is mandated. However, it is strongly recommended that the method be independent of drawing order, and fast enough to operate at interactive frame rates. If the chosen method requires post-processing per frame, that can be done in `releaseTarget`.

- **Dithering**. Specifies that dithering should be used to increase the perceived color depth of the screen. No particular method of dithering is mandated. However, the method should be optimized for animated content (as opposed to still images) and be able to operate at interactive frame rates. For example, ordered dithering is recommended over error diffusion. Implementations may choose to do the dithering at rendering time (per pixel) or as a post-process (upon `releaseTarget`).

- **True color rendering**. Specifies that rendering should be done with an internal color depth higher than what is supported by the device. For example, on a device with an RGB565 display, rendering could be done into an RGBA8 back buffer, truncating the pixels to 16 bits only upon `releaseTarget`. True color rendering is especially useful when combined with dithering.

The fact that a hint is supported does not guarantee that it is supported for all different types of rendering targets. For example, antialiasing may be supported for Image targets, but not Canvas targets. Implementations must indicate support for a hint if that hint is supported for at least one type of rendering target. Furthermore, if more than one hint is supported, it is not guaranteed that those hints can be used together. For example, antialiasing may preclude dithering, and vice versa. If the application specifies two or more hints, and that combination is not supported, the implementation may enable any one (or two) of those hints.

It is only meaningful for the implementation to support a hint if that allows the application to trade performance for quality or vice versa. For example, if dithering is built into the display hardware and is always enabled, the implementation should not indicate support for the dithering hint. Similarly, if the device has a 24 bpp display, the implementation should not support the true color hint (unless, of course, it uses even higher color precision in the back buffer).

## Implementation guidelines

See the package description for general implementation requirements, definitions of coordinate systems, and other background information.

### Depth buffer

The format and bit depth of the depth buffer are specific to each implementation and are not known to the application. However, the depth buffer resolution must be at least 8 bits. The contents of the depth buffer are never exposed through the API, so implementations may choose not to use a conventional depth buffer at all. However, a conforming implementation is required to behave *as if* it were using an ordinary depth buffer.

### Color buffer

The resolution of each color channel (R, G, B and A) in the color buffer must be at least 1 bit. Not all color channels are necessarily present; for example, the alpha channel is missing from Canvas and Image targets in MIDP. On a device with a black and white display, there may be only one channel, representing the luminance. In such a situation, the conversion of RGB colors into luminance can be done at any point in the pipeline, provided that the conversion is done according to the general rules set forth in the package description.

The `clear` and `render(World)` methods impose the restriction that the background image must be in the same format as the bound rendering target. It is worth highlighting that when bound to a MIDP Graphics object, the effective format can only be RGB (never RGBA) due to restrictions in the MIDP specification. It is also true that the MIDP Graphics object appears to be an RGB target even when the physical display is in fact monochrome. This reduces the complexity of application development considerably, since an RGB format image is valid for any binding to a MIDP target. Other target platforms may or may not be similarly specified.

### Back buffer

It is intentionally unspecified whether a separate back buffer should be allocated for colors or not. Instead, we leave it the implementation to decide which mode of operation is the most efficient, or which produces the best quality, according to the screen dimensions and speed versus quality preferences given by the application.

The decision whether to allocate any back buffer memory should be made at the latest when a new rendering target is first bound to the Graphics3D object. A previously bound rendering target is considered to be "new" when the rendering quality hints or the dimensions of the clipping rectangle have changed. In the case of a Graphics target, the actual rendering target is considered to be the Canvas, Image or other surface that the Graphics is attached to. The motivation for this rule is to guarantee that repeated rebinding of a rendering target - or several different rendering targets - will not incur the performance penalty of reallocating back buffer memory.

As an example of when a back buffer may be desired, consider a case where the application specifies `setDitheringEnable(true)` and subsequently binds a Canvas target. If the MIDP native color format is of low precision (such as RGB444), the implementation may wish to render at a higher color precision to a back buffer, then dither down to the MIDP native color format.

**Example:**
**A code fragment illustrating the usage of Graphics3D.**

```
public class MyCanvas extends Canvas
{
    World myWorld;
    int currentTime = 0;

    public MyCanvas() throws IOException {

        // Load an entire World. Proper exception handling is omitted
        // for clarity; see the class description of Loader for a more
        // elaborate example.

        Object3D[] objects = Loader.load("http://www.example.com/myscene.m3g");
        myWorld = (World) objects[0];
    }

    // The paint method is called by MIDP after the application has issued
    // a repaint request. We draw a new frame on this Canvas by binding the
    // current Graphics object as the target, then rendering, and finally
    // releasing the target.

    protected void paint(Graphics g) {

        // Get the singleton Graphics3D instance that is associated
        // with this midlet.

        Graphics3D g3d = Graphics3D.getInstance();

        // Bind the 3D graphics context to the given MIDP Graphics
        // object. The viewport will cover the whole of this Canvas.

        g3d.bindTarget(g);

        // Apply animations, render the scene and release the Graphics.

        myWorld.animate(currentTime);
        g3d.render(myWorld);    // render a view from the active camera
        g3d.releaseTarget();    // flush the rendered image
        currentTime += 50;      // assume we can handle 20 frames per second
    }
}
```

| **Field Summary** | |
|---|---|
| static int | **ANTIALIAS**<br>A parameter to `bindTarget`, specifying that antialiasing should be turned on. |
| static int | **DITHER**<br>A parameter to `bindTarget`, specifying that dithering should be turned on. |
| static int | **OVERWRITE**<br>A parameter to `bindTarget`, specifying that the existing contents of the rendering target need not be preserved. |
| static int | **TRUE_COLOR**<br>A parameter to `bindTarget`, specifying that true color rendering should be turned on. |

| **Method Summary** | |
|---|---|
| int | **addLight**(Light light, Transform transform)<br>Binds a Light to use in subsequent immediate mode rendering. |
| void | **bindTarget**(java.lang.Object target)<br>Binds the given Graphics or mutable Image2D as the rendering target of this Graphics3D. |
| void | **bindTarget**(java.lang.Object target, boolean depthBuffer, int hints)<br>Binds the given Graphics or mutable Image2D as the rendering target of this Graphics3D. |
| void | **clear**(Background background)<br>Clears the viewport as specified in the given Background object. |
| Camera | **getCamera**(Transform transform)<br>Returns the current camera. |
| float | **getDepthRangeFar**()<br>Returns the far distance of the depth range. |
| float | **getDepthRangeNear**()<br>Returns the near distance of the depth range. |
| int | **getHints**()<br>Returns the rendering hints given for the current rendering target. |
| static Graphics3D | **getInstance**()<br>Retrieves the singleton Graphics3D instance that is associated with this application. |
| Light | **getLight**(int index, Transform transform)<br>Returns a light in the current light array. |
| int | **getLightCount**()<br>Returns the size of the current light array. |
| static java.util.Hashtable | **getProperties**()<br>Retrieves implementation specific properties. |
| java.lang.Object | **getTarget**()<br>Returns the current rendering target. |

| | |
|---|---|
| int | **getViewportHeight**()<br>Returns the height of the viewport. |
| int | **getViewportWidth**()<br>Returns the width of the viewport. |
| int | **getViewportX**()<br>Returns the horizontal position of the viewport. |
| int | **getViewportY**()<br>Returns the vertical position of the viewport. |
| boolean | **isDepthBufferEnabled**()<br>Queries whether depth buffering is enabled for the current rendering target. |
| void | **releaseTarget**()<br>Flushes the rendered 3D image to the currently bound target and then releases the target. |
| void | **render**(Node node, Transform transform)<br>Renders the given Sprite3D, Mesh, or Group node with the given transformation from local coordinates to world coordinates. |
| void | **render**(VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform)<br>Renders the given submesh with the given transformation from local coordinates to world coordinates. |
| void | **render**(VertexBuffer vertices, IndexBuffer triangles, Appearance appearance, Transform transform, int scope)<br>Renders the given submesh with the given scope and the given transformation from local coordinates to world coordinates. |
| void | **render**(World world)<br>Renders an image of `world` as viewed by the active camera of that World. |
| void | **resetLights**()<br>Clears the array of current Lights. |
| void | **setCamera**(Camera camera, Transform transform)<br>Sets the Camera to use in subsequent immediate mode rendering. |
| void | **setDepthRange**(float near, float far)<br>Specifies the mapping of depth values from normalized device coordinates to window coordinates. |
| void | **setLight**(int index, Light light, Transform transform)<br>Replaces or modifies a Light currently bound for immediate mode rendering. |
| void | **setViewport**(int x, int y, int width, int height)<br>Specifies a rectangular viewport on the currently bound rendering target. |

## Field Detail

### ANTIALIAS

`public static final int` **`ANTIALIAS`**

A parameter to `bindTarget`, specifying that antialiasing should be turned on. The application may query from `getProperties` whether this hint is acted upon by the implementation.

**See Also:**
Constant Field Values

### DITHER

`public static final int` **`DITHER`**

A parameter to `bindTarget`, specifying that dithering should be turned on. The application may query from `getProperties` whether this hint is acted upon by the implementation.

**See Also:**
Constant Field Values

### TRUE_COLOR

`public static final int` **`TRUE_COLOR`**

A parameter to `bindTarget`, specifying that true color rendering should be turned on. The application may query from `getProperties` whether this hint is acted upon by the implementation.

**See Also:**
Constant Field Values

### OVERWRITE

`public static final int` **`OVERWRITE`**

A parameter to `bindTarget`, specifying that the existing contents of the rendering target need not be preserved. This can improve performance in applications that fully overwrite the rendering target without necessarily clearing it first.

**Since:**
M3G 1.1
**See Also:**
Constant Field Values

## Method Detail

### getInstance

`public static final Graphics3D` **`getInstance`**`()`

Retrieves the singleton Graphics3D instance that is associated with this application. The same instance will be returned every time.

Initially, the state of the Graphics3D instance is as follows:

- viewport : undefined (reset at `bindTarget`)
- depth range : [0, 1]
- current camera : none
- current lights : none

**Returns:**
the Graphics3D instance associated with this application

## bindTarget

`public void **bindTarget**(java.lang.Object target)`

Binds the given Graphics or mutable Image2D as the rendering target of this Graphics3D. The type of the Graphics object depends on the Java profile that this specification is implemented on, as follows:

- `java.awt.Graphics` on profiles supporting AWT;
- `javax.microedition.lcdui.Graphics` on profiles supporting LCDUI;
- either of the above on profiles supporting both AWT and LCDUI.

The state of this Graphics3D after calling this method will be as follows:

- rendering target : the given Graphics or Image2D
- viewport : covering the target clipping rectangle
- depth buffer : enabled
- antialiasing : disabled
- dithering : disabled
- true color : disabled
- overwrite : disabled
- depth range : as before
- current camera : as before
- current lights : as before

The dimensions of the given target must not exceed the implementation-specific maximum viewport size, which can be queried with `getProperties`. The viewport is set such that its top left corner is at the top left corner of the target clipping rectangle, and its dimensions are equal to those of the clipping rectangle.

Note that this method will not block waiting if another thread has already bound a rendering target to this Graphics3D. Instead, it will throw an exception. Only one target can be bound at a time, and it makes no difference whether that target has been bound from the current thread or some other thread.

**Parameters:**
`target` - the Image2D or Graphics object to receive the rendered image
**Throws:**
`java.lang.NullPointerException` - if `target` is null
`java.lang.IllegalStateException` - if this Graphics3D already has a rendering target
`java.lang.IllegalArgumentException` - if `target` is not a mutable Image2D object or a

Graphics object appropriate to the underlying Java profile
`java.lang.IllegalArgumentException` - if `(target.width > maxViewportWidth) || (target.height > maxViewportHeight)`
`java.lang.IllegalArgumentException` - if `target` is an Image2D with an internal format other than `RGB` or `RGBA`

**See Also:**
`releaseTarget`, `getHints`, `isDepthBufferEnabled`

### bindTarget

```
public void bindTarget(java.lang.Object target,
                       boolean depthBuffer,
                       int hints)
```

Binds the given Graphics or mutable Image2D as the rendering target of this Graphics3D. This method is identical to the simpler variant of `bindTarget`, but allows the depth buffering enable flag and the rendering hints to be specified. See the class description for more information on these.

If the depth buffer is disabled, depth testing and depth writing are implicitly disabled for *all* objects, regardless of their individual CompositingMode settings.

**Parameters:**
`target` - the Image2D or Graphics object to receive the rendered image
`depthBuffer` - *true* to enable depth buffering; *false* to disable
`hints` - an integer bitmask specifying which rendering hints to enable, or zero to disable all hints

**Throws:**
`java.lang.NullPointerException` - if `target` is null
`java.lang.IllegalStateException` - if this Graphics3D already has a rendering target
`java.lang.IllegalArgumentException` - if `target` is not a mutable Image2D object or a Graphics object appropriate to the underlying Java profile
`java.lang.IllegalArgumentException` - if `target` is an Image2D with an internal format other than `RGB` or `RGBA`
`java.lang.IllegalArgumentException` - if `(target.width > maxViewportWidth) || (target.height > maxViewportHeight)`
`java.lang.IllegalArgumentException` - if `hints` is not zero or an OR bitmask of one or more of `ANTIALIAS, DITHER, TRUE_COLOR`, and `OVERWRITE`

**See Also:**
`releaseTarget`, `getHints`, `isDepthBufferEnabled`

### releaseTarget

```
public void releaseTarget()
```

Flushes the rendered 3D image to the currently bound target and then releases the target. This ensures that the 3D image is actually made visible on the target that was set in `bindTarget`. Otherwise, the image may or may not become visible. If no target is bound, the request to release the target is silently ignored.

**See Also:**
`bindTarget`

### getTarget

```
public java.lang.Object getTarget()
```

Returns the current rendering target.

**Returns:**
the currently bound rendering target, or `null` if no target is bound

**Since:**
M3G 1.1

**See Also:**
`bindTarget`

## getHints

```
public int getHints()
```

Returns the rendering hints given for the current rendering target. If no target is bound, the return value is undefined.

Note that the return value is the hint bitmask set by the application, even if the implementation is not acting upon all of the hints in it.

**Returns:**
the current rendering hint bitmask

**Since:**
M3G 1.1

**See Also:**
`bindTarget`

## isDepthBufferEnabled

```
public boolean isDepthBufferEnabled()
```

Queries whether depth buffering is enabled for the current rendering target. If no target is bound, the return value is undefined.

**Returns:**
`true` if depth buffering is enabled, `false` if not

**Since:**
M3G 1.1

**See Also:**
`bindTarget`

## setViewport

```
public void setViewport(int x,
                        int y,
                        int width,
                        int height)
```

Specifies a rectangular viewport on the currently bound rendering target. The viewport is the area where the view of the current camera will appear. Any parts of the viewport that lie outside the boundaries of the target clipping rectangle are silently clipped off; however, this must simply discard the pixels without affecting projection. The viewport upper left corner (x, y) is given relative to the origin for a Graphics rendering target, or the upper left corner for an Image2D target. Refer to the class description for details.

The viewport mapping transforms vertices from normalized device coordinates ($x_{ndc}$, $y_{ndc}$) to window coordinates ($x_w$, $y_w$) as follows:

$$x_w = 0.5\ x_{ndc} w + o_x$$
$$y_w = -0.5\ y_{ndc} h + o_y$$

where w and h are the width and height of the viewport, specified in pixels, and ($o_x$, $o_y$) is the center of the viewport, also in pixels. The center of the viewport is obtained from the (x, y) coordinates of the top left corner as follows:

$$o_x = x + 0.5\ w$$
$$o_y = y + 0.5\ h$$

**Parameters:**

`x` - X coordinate of the viewport upper left corner, in pixels
`y` - Y coordinate of the viewport upper left corner, in pixels
`width` - width of the viewport, in pixels
`height` - height of the viewport, in pixels

**Throws:**

`java.lang.IllegalArgumentException` - if `(width <= 0) || (height <= 0)` (note that `x` and `y` may have any value)
`java.lang.IllegalArgumentException` - if `(width > maxViewportWidth) || (height > maxViewportHeight)`

**See Also:**

`bindTarget`, `getViewportX`, `getViewportY`, `getViewportWidth`, `getViewportHeight`

## getViewportX

`public int getViewportX()`

Returns the horizontal position of the viewport.

**Returns:**

the X coordinate of the upper left corner, in pixels

**Since:**

M3G 1.1

**See Also:**

`setViewport`

## getViewportY

`public int getViewportY()`

Returns the vertical position of the viewport.

**Returns:**
the Y coordinate of the upper left corner, in pixels

**Since:**
M3G 1.1

**See Also:**
`setViewport`

## getViewportWidth

```
public int getViewportWidth()
```

Returns the width of the viewport.

**Returns:**
the width of the viewport, in pixels

**Since:**
M3G 1.1

**See Also:**
`setViewport`

## getViewportHeight

```
public int getViewportHeight()
```

Returns the height of the viewport.

**Returns:**
the height of the viewport, in pixels

**Since:**
M3G 1.1

**See Also:**
`setViewport`

## setDepthRange

```
public void setDepthRange(float near,
                          float far)
```

Specifies the mapping of depth values from normalized device coordinates to window coordinates. Window coordinates are used for depth buffering.

Depth values may range from -1 to 1 in normalized device coordinates (NDC), and from 0 to 1 in window coordinates. By default, the whole [0, 1] range of window coordinates is used. This method allows the normalized device coordinates [-1, 1] to be mapped to a "tighter" interval of window coordinates, for example, (0.5, 1].

Formally, the Z coordinate of a vertex in NDC, $z_{ndc}$, is transformed to window coordinates ($z_w$) as follows:

$z_w = 0.5\ (\text{far} - \text{near})\ (z_{ndc} + 1) + \text{near}$

where `near` and `far` are the distances, in window coordinates, to the near and far plane of the depth range, respectively. Both distances must be in [0, 1]. However, it is not necessary for the near plane to be closer than the far plane; inverse mappings are also acceptable.

**Parameters:**
`near` - distance to the near clipping plane, in window coordinates
`far` - distance to the far clipping plane, in window coordinates

**Throws:**
`java.lang.IllegalArgumentException` - if `(near < 0) || (near > 1)`
`java.lang.IllegalArgumentException` - if `(far < 0) || (far > 1)`

**See Also:**
`getDepthRangeNear`, `getDepthRangeFar`

## getDepthRangeNear

`public float` **`getDepthRangeNear`**`()`

Returns the near distance of the depth range.

**Returns:**
distance to the near clipping plane, in window coordinates

**Since:**
M3G 1.1

**See Also:**
`setDepthRange`

## getDepthRangeFar

`public float` **`getDepthRangeFar`**`()`

Returns the far distance of the depth range.

**Returns:**
distance to the far clipping plane, in window coordinates

**Since:**
M3G 1.1

**See Also:**
`setDepthRange`

## clear

`public void` **`clear`**`(Background background)`

Clears the viewport as specified in the given Background object. If the background object is null, the default settings are used. That is, the color buffer is cleared to transparent black, and the depth buffer to the maximum depth value (1.0).

**Parameters:**

`background` - a Background object defining which buffers to clear and how, or null to use the default settings

**Throws:**

`java.lang.IllegalArgumentException` - if the background image in `background` is not in the same format as the currently bound rendering target
`java.lang.IllegalStateException` - if this Graphics3D does not have a rendering target

## render

```
public void render(World world)
```

Renders an image of `world` as viewed by the active camera of that World. The node transformation of the World is ignored, but its other attributes are respected.

Contrary to the immediate mode `render` variants, this method automatically clears the color buffer and the depth buffer according to the Background settings of the World.

Prior to rendering, the current camera and lights set in this Graphics3D are automatically overwritten with the active camera and lights of the World. Upon method return, the lights array will contain precisely those Light nodes whose effective rendering enable flag is *true* (see `Node.setRenderingEnable`). The Lights are written to the array in undefined order, but such that the first Light is at index 0 and there are no empty slots interleaved within non-empty slots. As usual, the Camera and Light transformations will be from their local coordinates to world coordinates (i.e., the coordinate system of `world`). In other words, the Camera and Lights are effectively set up as follows:

```
Camera c = world.getActiveCamera();
myG3D.setCamera(c, c.getTransformTo(world));
myG3D.resetLights();
for (<all enabled Lights l in world>) {
   myG3D.addLight(l, l.getTransformTo(world));
}
```

For any node that is rendered, if the transformation from that node's local coordinates to the camera space is not invertible, the results of lighting and fogging are undefined.

**Parameters:**

`world` - the World to render

**Throws:**

`java.lang.NullPointerException` - if `world` is null
`java.lang.IllegalStateException` - if this Graphics3D does not have a rendering target
`java.lang.IllegalStateException` - if `world` has no active camera, or the active camera is not in that world
`java.lang.IllegalStateException` - if the background image of `world` is not in the same format as the currently bound rendering target
`java.lang.IllegalStateException` - if any Mesh that is rendered violates the constraints defined in Mesh, MorphingMesh, SkinnedMesh, VertexBuffer, or IndexBuffer
`java.lang.ArithmeticException` - if the transformation from the active camera of `world` to the world space is uninvertible

## render

```
public void render(Node node,
                   Transform transform)
```

Renders the given Sprite3D, Mesh, or Group node with the given transformation from local coordinates to world coordinates. The node transformation of the given node is ignored, but its other attributes are respected.

Any ancestors of the given node are ignored, as well as their transformations and other attributes. The node's descendants, if any, are rendered as usual. However, any Camera and Light nodes among the descendants are ignored and the camera and lights of this Graphics3D are used instead.

The scope masks of the current Lights and Camera are respected, as well as the rendering enable flags of the Lights. The rendering enable flag of the Camera is ignored, as always.

Note that Mesh nodes include MorphingMesh and SkinnedMesh nodes, and that Group nodes include World nodes. If a World is passed to this method, it is simply treated like any other Group and therefore any Background, Camera and Light objects it may have are ignored.

This method does *not* clear the color and depth buffers; the application must explicitly clear them with the `clear` method and/or draw any background graphics beforehand.

For any node that is rendered, if the transformation from that node's local coordinates to the camera space is not invertible, the results of lighting and fogging are undefined.

**Parameters:**

`node` - the Sprite3D, Mesh, or Group to render
`transform` - the transformation from the local coordinate system of `node` to world space, or null to indicate the identity matrix

**Throws:**

`java.lang.NullPointerException` - if `node` is null
`java.lang.IllegalArgumentException` - if `node` is not a Sprite3D, Mesh, or Group
`java.lang.IllegalStateException` - if this Graphics3D does not have a rendering target
`java.lang.IllegalStateException` - if this Graphics3D does not have a current camera
`java.lang.IllegalStateException` - if any Mesh that is rendered violates the constraints defined in Mesh, MorphingMesh, SkinnedMesh, VertexBuffer, or IndexBuffer

## render

```
public void render(VertexBuffer vertices,
                   IndexBuffer triangles,
                   Appearance appearance,
                   Transform transform,
                   int scope)
```

Renders the given submesh with the given scope and the given transformation from local coordinates to world coordinates.

The scope masks of the current Lights and Camera are respected, as well as the rendering enable flags of the Lights. The rendering enable flag of the Camera is ignored, as always.

If the transformation from local coordinates to the camera space is not invertible, the results of lighting and fogging are undefined.

**Parameters:**

`vertices` - a VertexBuffer defining the vertex attributes
`triangles` - an IndexBuffer defining the triangle strips
`appearance` - an Appearance defining the surface properties
`transform` - the transformation from the local coordinate system of `vertices` to world space, or null to indicate the identity matrix
`scope` - the scope of the submesh; this determines whether the submesh is rendered at all, and if it is, which lights are used; "-1" makes the scope as wide as possible

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `triangles` is null
`java.lang.NullPointerException` - if `appearance` is null
`java.lang.IllegalStateException` - if this Graphics3D does not have a rendering target
`java.lang.IllegalStateException` - if this Graphics3D does not have a current camera
`java.lang.IllegalStateException` - if `vertices` or `triangles` violates the constraints defined in VertexBuffer or IndexBuffer

## render

```
public void render(VertexBuffer vertices,
                   IndexBuffer triangles,
                   Appearance appearance,
                   Transform transform)
```

Renders the given submesh with the given transformation from local coordinates to world coordinates. This method is exactly the same as the other submesh rendering method, except that the scope is implicitly set to -1 (the widest possible).

**Parameters:**

`vertices` - a VertexBuffer defining the vertex attributes
`triangles` - an IndexBuffer defining the triangle strips
`appearance` - an Appearance defining the surface properties
`transform` - the transformation from the local coordinate system of `vertices` to world space, or null to indicate the identity matrix

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `triangles` is null
`java.lang.NullPointerException` - if `appearance` is null
`java.lang.IllegalStateException` - if this Graphics3D does not have a rendering target
`java.lang.IllegalStateException` - if this Graphics3D does not have a current camera
`java.lang.IllegalStateException` - if `vertices` or `triangles` violates the constraints defined in VertexBuffer or IndexBuffer

## setCamera

```
public void setCamera(Camera camera,
                      Transform transform)
```

Sets the Camera to use in subsequent immediate mode rendering. The given transformation is from the Camera's local coordinate system (camera space) to the world space. The transformation is copied in, so any further changes to it will not be reflected in this Graphics3D. The node transformation of the Camera is ignored. If the Camera has any ancestors, they are also ignored.

The scope of the Camera is respected when rendering. The rendering enable flag of the Camera is ignored, as always.

The given camera-to-world transformation must be invertible in order that the model-to-camera (or "modelview") transformation for each rendered object and light source can be computed.

**Parameters:**

`camera` - the Camera to bind for immediate mode rendering, or null to unbind the current camera
`transform` - the transformation from the local coordinate system of `camera` to world space, or null to indicate the identity matrix

**Throws:**

`java.lang.ArithmeticException` - if `transform` is not invertible

**See Also:**

`getCamera`

## getCamera

```
public Camera getCamera(Transform transform)
```

Returns the current camera.

**Parameters:**

`transform` - a Transform to store the current camera transformation in, or `null` to only get the camera

**Returns:**

the current camera

**Since:**

M3G 1.1

**See Also:**

`setCamera`

## addLight

```
public int addLight(Light light,
                    Transform transform)
```

Binds a Light to use in subsequent immediate mode rendering. The Light is inserted at the end of the current lights array, regardless of whether there are empty (null) slots at lower indices. The index of the slot in which the Light is inserted is returned to the application. The returned indices are guaranteed to be strictly increasing, until `render(World)` or `resetLights` is called.

The given transformation is from the Light's local coordinate system to the world space. Note that the transformation need not be invertible. The transformation is copied in, so any further changes to it will not be reflected in this Graphics3D. The node transformation of the Light is ignored. If the Light has any ancestors, they are also ignored.

Note that the complete transformation is not required for performing lighting computations. Implementations may therefore opt to store only the required parts of it.

The index of the added Light is guaranteed to remain the same until the light is either removed using `setLight`, or the lights array is overwritten by `render(World)`, or the array is explicitly cleared with `resetLights`.

The scope and rendering enable flag of the Light are respected when rendering. This enables selection of subsets of the current light array without repeatedly resetting the light array.

**Parameters:**

`light` - the Light to add at the end of the array of current lights
`transform` - the transformation from the local coordinate system of `light` to world space, or null to indicate the identity matrix

**Returns:**

the index at which the Light was inserted in the array

**Throws:**

`java.lang.NullPointerException` - if `light` is null

**See Also:**

`setLight`, `getLight`

## setLight

```
public void setLight(int index,
                     Light light,
                     Transform transform)
```

Replaces or modifies a Light currently bound for immediate mode rendering. This is similar to addLight, except that an existing light is replaced and the size of the light array does not change.

**Parameters:**

`index` - index of the light to set
`light` - the Light to set, or null to remove the light at `index`
`transform` - the transformation from the local coordinate system of `light` to world space, or null to indicate the identity matrix

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getLightCount)`

**See Also:**

`addLight`, `getLight`

## resetLights

```
public void resetLights()
```

Clears the array of current Lights.

## getLightCount

```
public int getLightCount()
```

Returns the size of the current light array. This includes actual lights as well as any empty slots in the array.

**Returns:**
the number of slots in the current light array

**Since:**
M3G 1.1

**See Also:**
`addLight`, `setLight`, `getLight`

## getLight

```
public Light getLight(int index,
                      Transform transform)
```

Returns a light in the current light array.

Note that implementations are not required to store the complete light transformation passed in `addLight` or `setLight`. The returned transformation may therefore be different from the transformation passed in, but must produce the same lighting.

**Parameters:**
`index` - index of the light to get
`transform` - transform to store the light transformation in, or `null` to only get the Light object

**Returns:**
the light object at `index`

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getLightCount)`

**Since:**
M3G 1.1

**See Also:**
`addLight`, `setLight`, `getLightCount`

## getProperties

```
public static final java.util.Hashtable getProperties()
```

Retrieves implementation specific properties. The properties are stored in a Hashtable that is keyed by String values. The Hashtable will always contain the entries listed in the table below, but there may also be other entries specific to each implementation.

The third column shows for each property the baseline requirement that all implementations must satisfy. The actual value returned may be equal to or greater than the baseline requirement.

| Key (String) | Value type | Minimum requirement | Description |
|---|---|---|---|
| supportAntialiasing | Boolean | false | See above |
| supportTrueColor | Boolean | false | See above |
| supportDithering | Boolean | false | See above |

| | | | |
|---|---|---|---|
| supportMipmapping | Boolean | false | See Texture2D |
| supportPerspectiveCorrection | Boolean | false | See PolygonMode |
| supportLocalCameraLighting | Boolean | false | See PolygonMode |
| maxLights | Integer | 8 | See Light |
| maxViewportWidth | Integer | 256 | See setViewport |
| maxViewportHeight | Integer | 256 | See setViewport |
| maxViewportDimension | Integer | 256 | The minimum of {maxViewportWidth, maxViewportHeight} |
| maxTextureDimension | Integer | 256 | See Texture2D |
| maxSpriteCropDimension | Integer | 256 | See Sprite3D |
| maxTransformsPerVertex | Integer | 2 | See SkinnedMesh |
| numTextureUnits | Integer | 1 | See Appearance |

**Returns:**

a Hashtable defining properties specific to this implementation

**javax.microedition.m3g**

# Class Group

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
          └ javax.microedition.m3g.Node
              └ javax.microedition.m3g.Group
```

**Direct Known Subclasses:**
World

public class **Group**
extends Node

A scene graph node that stores an unordered set of nodes as its children.

The parent-child relationship is bidirectional in the sense that if node A is a child of node B, then B is the (one and only) parent of A. In particular, the `getParent` method of A will return B. Besides Group nodes, this also concerns SkinnedMesh nodes: the skeleton group is the one and only child of a SkinnedMesh.

A node can have at most one parent at a time, and cycles are prohibited. Furthermore, a World node cannot be a child of any node. These rules are enforced by the `addChild` method in this class, as well as the constructor of SkinnedMesh.

**See Also:**
Binary format

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| **`Group`**`()`<br>Constructs a new Group node and initializes it with an empty list of children. |

## Method Summary

| | |
|---|---|
| `void` | **`addChild`**`(Node child)`<br>Adds the given node to this Group, potentially changing the order and indices of the previously added children. |
| `Node` | **`getChild`**`(int index)`<br>Gets a child by index. |

| | |
|---|---|
| int | **getChildCount**() <br> Gets the number of children in this Group. |
| boolean | **pick**(int scope, float x, float y, Camera camera, RayIntersection ri) <br> Picks the first Mesh or scaled Sprite3D in this Group that is enabled for picking, is intercepted by the given pick ray, and is in the specified scope. |
| boolean | **pick**(int scope, float ox, float oy, float oz, float dx, float dy, float dz, RayIntersection ri) <br> Picks the first Mesh in this Group that is intercepted by the given pick ray and is in the specified scope. |
| void | **removeChild**(Node child) <br> Removes the given node from this Group, potentially changing the order and indices of the remaining children. |

| **Methods inherited from class javax.microedition.m3g.Node** |
|---|
| align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject |

## Constructor Detail

### Group

`public Group()`

Constructs a new Group node and initializes it with an empty list of children. Properties inherited from Object3D and Node will have the default values as specified in their respective class descriptions.

## Method Detail

### addChild

`public void addChild(Node child)`

Adds the given node to this Group, potentially changing the order and indices of the previously added children. The position at which the node is inserted among the existing children is deliberately left undefined. This gives implementations the freedom to select a data structure that best fits their needs, instead of mandating a particular kind of data structure.

**Parameters:**
`child` - the node to add; must not form a loop in the scene graph

**Throws:**
`java.lang.NullPointerException` - if `child` is null
`java.lang.IllegalArgumentException` - if `child` is this Group
`java.lang.IllegalArgumentException` - if `child` is a World node
`java.lang.IllegalArgumentException` - if `child` already has a parent other than this Group
`java.lang.IllegalArgumentException` - if `child` is an ancestor of this Group

## removeChild

`public void removeChild(Node child)`

Removes the given node from this Group, potentially changing the order and indices of the remaining children. If the given node is not a child of this Group, or is null, the request to remove it is silently ignored.

**Parameters:**
`child` - the node to remove

**Throws:**
`java.lang.IllegalArgumentException` - if removing `child` would break a connection between a SkinnedMesh node and one of its transform references

## getChildCount

`public int getChildCount()`

Gets the number of children in this Group.

**Returns:**
the number of children directly attached to this group

## getChild

`public Node getChild(int index)`

Gets a child by index. Valid indices range from zero up to the number of children minus one. Note that the index of any child may change whenever a node is added to or removed from this Group. See `addChild` for more information.

**Parameters:**
`index` - index of the child node to get

**Returns:**

the child node at the given index; can not be null

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getChildCount)`

## pick

```
public boolean pick(int scope,
                    float ox,
                    float oy,
                    float oz,
                    float dx,
                    float dy,
                    float dz,
                    RayIntersection ri)
```

Picks the first Mesh in this Group that is intercepted by the given pick ray and is in the specified scope. Meshes that are disabled or out of scope are ignored. Any ancestors of this Group, including their picking enable flags, are ignored. Winding and culling flags for each Mesh are respected when determining a hit, such that triangles culled based on their facing with respect to the pick ray are ignored.

The pick ray is cast in the given direction from the given location in the coordinate system of this Group. The direction vector of the ray does not need to be unit length; the distance to the picked object is computed relative to the length of the given ray.

Information about the picked object, if any, is filled in to the given RayIntersection object. If no intersection occurs, the RayIntersection object is left unmodified.

This method ignores all Sprite3D nodes. This is because the camera parameters (that is, the projection matrix) are required in order to compute the size of a sprite (see the Sprite3D class description), and that information is not available to this method. Developers are advised to use the other `pick` variant if picking of sprites is desired.

The application should ensure that there are no uninvertible node transformations in this Group. Depending on how picking is implemented, singular transformations may or may not trigger an ArithmeticException.

**Parameters:**

`scope` - an integer scope specifying which Meshes to test for intersection with the pick ray; "-1" makes the scope as wide as possible
`ox` - X coordinate of the ray origin
`oy` - Y coordinate of the ray origin
`oz` - Z coordinate of the ray origin
`dx` - X component of the ray direction
`dy` - Y component of the ray direction
`dz` - Z component of the ray direction
`ri` - a RayIntersection object to fill in with information about the intersected Mesh, or null to just find out whether the ray intersected something or not

**Returns:**

*true* if the ray intersected a Mesh; *false* otherwise

**Throws:**

`java.lang.IllegalArgumentException` - if `dx = dy = dz = 0`
`java.lang.IllegalStateException` - if any Mesh that is tested for intersection violates the constraints defined in Mesh, MorphingMesh, SkinnedMesh, VertexBuffer, or IndexBuffer

`java.lang.ArithmeticException` - if the inverse of an uninvertible transformation is required by the implementation

## pick

```
public boolean pick(int scope,
                    float x,
                    float y,
                    Camera camera,
                    RayIntersection ri)
```

Picks the first Mesh or scaled Sprite3D in this Group that is enabled for picking, is intercepted by the given pick ray, and is in the specified scope.

This method behaves identically to the other `pick` variant, except that the pick ray is specified differently and that scaled sprites can also be picked. Unscaled sprites can not be picked. This is because the size of an unscaled sprite is only defined in screen space (that is, after viewport transformation), and the viewport parameters are not available to this method. See the Sprite3D class description for more information on sprite picking.

The pick ray is cast from the given point **p** = (x, y) on the near clipping plane towards the corresponding point on the far clipping plane, and then beyond. See the Implementation guidelines below for details.

Note that the origin of the pick ray is *not* the given Camera, but the point on the near clipping plane. Consequently the distance to the picked object, returned in RayIntersection, is not the distance from the camera, but the distance from the point **p**.

The point **p** is specified relative to the viewport such that (0, 0) is the upper left corner and (1, 1) is the lower right corner. However, the (x, y) coordinates are not restricted to that range and may take on any values. In other words, objects that do not lie within the viewport can also be picked.

The given Camera and this Group must be in the same scene graph. Furthermore, the projection matrix of the Camera must be invertible. Depending on how picking is implemented, objects within the Group which have uninvertible modelview matrices may or may not trigger an ArithmeticException.

### Implementation guidelines

The pick ray is cast towards infinity from the given point **p** on the near clipping plane, through a point **p**' on the far clipping plane. The exact procedure of deriving the pick ray origin and direction from the given point (x, y) and the given projection matrix **P** is as follows.

In normalized device coordinates (NDC), the viewport spans the range [-1, 1] in each dimension (X, Y and Z). Points that lie on the near plane have a Z coordinate of -1 in NDC; points on the far plane have a Z of 1. The normalized device coordinates of **p** and **p**' are, therefore:

$$\mathbf{p}_{ndc} = (2x-1,\ 1-2y,\ -1,\ 1)^T$$
$$\mathbf{p}'_{ndc} = (2x-1,\ 1-2y,\ 1,\ 1)^T$$

Note that the Y coordinate is inverted when going from NDC to viewport or vice versa, as the viewport upper left corner maps to (-1, 1) in NDC (see also the viewport transformation equation in `Graphics3D.setViewport`). Applying the inverse projection matrix on the pick points, we obtain their positions in camera space:

$$\mathbf{p}_c = \mathbf{P}^{-1}\mathbf{p}_{ndc}$$
$$\mathbf{p}'_c = \mathbf{P}^{-1}\mathbf{p}'_{ndc}$$

We then scale the resultant homogeneous points such that their W components are equal to 1; that might not otherwise be the case after the inverse projection. Formally, denoting the W components of the near and far points by w and w', the final camera space coordinates are obtained as follows:

$$\mathbf{p} = \mathbf{p}_c / w$$
$$\mathbf{p}' = \mathbf{p}'_c / w'$$

The origin of the pick ray in camera coordinates is then **p** while its direction vector is **p**' - **p**.

Finally, the pick ray is transformed from camera space to the coordinate system of this Group. That ray is used in the actual intersection tests, and is also the one that is returned by the `getRay` method in RayIntersection.

**Parameters:**

`scope` - an integer scope specifying which meshes and sprites to test for intersection with the pick ray; -1 makes the scope as wide as possible
`x` - X coordinate of the point on the viewport plane through which to cast the ray
`y` - Y coordinate of the point on the viewport plane through which to cast the ray
`camera` - a camera based on which the origin and direction of the pick ray are to be computed
`ri` - a RayIntersection object to fill in with information about the intersected Mesh, or null to just find out whether the ray intersected something or not

**Returns:**

*true* if the ray intersected a Mesh or Sprite3D; *false* otherwise

**Throws:**

`java.lang.NullPointerException` - if `camera` is null
`java.lang.IllegalStateException` - if any Mesh that is tested for intersection violates the constraints defined in Mesh, MorphingMesh, SkinnedMesh, VertexBuffer, or IndexBuffer
`java.lang.IllegalStateException` - if there is no scene graph path between `camera` and this Group
`java.lang.ArithmeticException` - if the inverse of an uninvertible transformation is required by the implementation

**javax.microedition.m3g**

## Class Image2D

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Image2D
```

public class **Image2D**
extends Object3D

A two-dimensional image that can be used as a texture, background or sprite image.

There are two kinds of images: mutable and immutable. The contents of a *mutable* image can be updated at any time by rendering into it or by using the `set` method. The changes are immediately reflected to the textures and other objects where the Image2D is used. The contents of an *immutable* image, on the other hand, are fixed at construction time and can not be changed later. Knowing that an image is immutable, the implementation can safely preprocess the image to optimize it for speed or memory consumption. For example, it may compress the image, reorder the pixels, or reduce the color depth.

The dimensions of the image are restricted only by the amount of available memory. However, if the image is to be used as a texture, its dimensions must be non-negative powers of two. This restriction is enforced by Texture2D.

The image contents are copied in from a byte array or from an Image object. All byte data supplied to an Image2D is treated as unsigned. That is, byte values in [-128, -1] are interpreted as values in [128, 255], in that order.

**See Also:**
Binary format

| Field Summary | |
|---|---|
| `static int` | **`ALPHA`**<br>A constructor parameter specifying that this Image2D has an alpha component only. |
| `static int` | **`LUMINANCE`**<br>A constructor parameter specifying that this Image2D has a luminance component only. |
| `static int` | **`LUMINANCE_ALPHA`**<br>A constructor parameter specifying that this Image2D has luminance and alpha components. |
| `static int` | **`RGB`**<br>A constructor parameter specifying that this Image2D has red, green and blue color components. |
| `static int` | **`RGBA`**<br>A constructor parameter specifying that this Image2D has red, green, blue and alpha components. |

| Constructor Summary |
|---|
| **`Image2D`**`(int format, int width, int height)`<br>Constructs an empty, mutable Image2D with the given dimensions. |

| |
|---|
| **Image2D**(int format, int width, int height, byte[] image)<br>Constructs an immutable Image2D by copying pixels from a byte array. |
| **Image2D**(int format, int width, int height, byte[] image, byte[] palette)<br>Constructs an immutable Image2D by copying palette indices from a byte array, and the palette entries from another byte array. |
| **Image2D**(int format, java.lang.Object image)<br>Constructs an immutable Image2D by copying pixels from a MIDP or AWT Image. |

## Method Summary

| | |
|---|---|
| int | **getFormat**()<br>Gets the internal format of this Image2D. |
| int | **getHeight**()<br>Gets the height of this Image2D, in pixels. |
| int | **getWidth**()<br>Gets the width of this Image2D, in pixels. |
| boolean | **isMutable**()<br>Queries whether this Image2D is mutable. |
| void | **set**(int x, int y, int width, int height, byte[] image)<br>Updates a rectangular area of this Image2D by copying pixels from a byte array. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject |

## Field Detail

### ALPHA

```
public static final int ALPHA
```

A constructor parameter specifying that this Image2D has an alpha component only. If the image data is supplied as a byte array, it must have one byte per pixel, representing the alpha value. This image format is useful for rendering objects with varying opacities, such as clouds. An ALPHA image can not be used as a rendering target or background.

**See Also:**
Constant Field Values

### LUMINANCE

```
public static final int LUMINANCE
```

A constructor parameter specifying that this Image2D has a luminance component only. If the image data is supplied as a byte array, it must have one byte per pixel, representing the luminance value. This image format is a cheaper alternative to RGB in cases where colors are not needed, for instance in light mapping. A `LUMINANCE` image can not be used as a rendering target or background.

**See Also:**
Constant Field Values

### LUMINANCE_ALPHA

```
public static final int LUMINANCE_ALPHA
```

A constructor parameter specifying that this Image2D has luminance and alpha components. If the image data is supplied as a byte array, it must have two bytes per pixel, representing the luminance and alpha values, in that order. This image format is a cheap alternative to RGBA in cases where colors are not needed; for instance in light mapping. A `LUMINANCE_ALPHA` image can not be used as a rendering target or background.

**See Also:**
Constant Field Values

### RGB

```
public static final int RGB
```

A constructor parameter specifying that this Image2D has red, green and blue color components. If the image data is supplied as a byte array, it must be in RGB order, one byte per component.

**See Also:**
Constant Field Values

### RGBA

```
public static final int RGBA
```

A constructor parameter specifying that this Image2D has red, green, blue and alpha components. If the image data is supplied as a byte array, it must be in RGBA order, one byte per component.

**See Also:**
Constant Field Values

## Constructor Detail

### Image2D

```
public Image2D(int format,
               java.lang.Object image)
```

Constructs an immutable Image2D by copying pixels from a MIDP or AWT Image. The type of the Image object depends on the Java profile that this specification is implemented on, as follows:

- `java.awt.Image` on profiles supporting AWT;
- `javax.microedition.lcdui.Image` on profiles supporting LCDUI;
- either of the above on profiles supporting both AWT and LCDUI.

The width and height of the created Image2D are set equal to those of the given Image. If the internal format of this Image2D is incompatible with the source Image, the pixels are converted to the internal format as they are copied in. The conversion is done according to the general rules that are specified in the package description. Note, in particular, that conversion from RGBA to ALPHA is done by copying in the alpha values as such and discarding the RGB values.

Because the internal pixel format of an Image is not exposed in MIDP, this method treats all mutable MIDP Images as RGB and all immutable Images as RGBA. The actual pixel format is respected on platforms such as AWT that do expose it to the application.

**Parameters:**
`format` - the internal pixel format of the new Image2D
`image` - pixels and image properties to copy into the new Image2D
**Throws:**
`java.lang.NullPointerException` - if `image` is null
`java.lang.IllegalArgumentException` - if `format` is not one of the symbolic constants listed above
`java.lang.IllegalArgumentException` - if `image` is not an Image object appropriate to the underlying Java profile

## Image2D

```
public Image2D(int format,
               int width,
               int height,
               byte[] image)
```

Constructs an immutable Image2D by copying pixels from a byte array.

Pixels in `image` are ordered from left to right and top to bottom. The number of bytes per pixel and the order of components are determined by the specified format; see above.

**Parameters:**
`format` - the internal pixel format of the new Image2D
`width` - width of the created image in pixels; must be positive
`height` - height of the created image in pixels; must be positive
`image` - pixel data as a byte array
**Throws:**
`java.lang.NullPointerException` - if `image` is null
`java.lang.IllegalArgumentException` - if `width <= 0`
`java.lang.IllegalArgumentException` - if `height <= 0`
`java.lang.IllegalArgumentException` - if `format` is not one of the symbolic constants listed above
`java.lang.IllegalArgumentException` - if `image.length < width*height*bpp`, where `bpp` is the number of bytes per pixel

## Image2D

```
public Image2D(int format,
               int width,
               int height,
               byte[] image,
               byte[] palette)
```

Constructs an immutable Image2D by copying palette indices from a byte array, and the palette entries from another byte array.

Pixels in `image` are ordered from left to right and top to bottom. Each pixel is composed of one byte, representing a palette index.

The palette consists of 256 entries, all with the same number of color components. The number and ordering of the color components is determined by the specified format; see the symbolic constants listed above. Note that a palette consisting of only alpha or luminance values is also allowed.

The palette entries are copied in from the `palette` array, starting at index 0. If the array has N entries, N < 256, the remaining entries [N, 255] are left undefined. If the array has more than 256 entries, only the first 256 are copied in.

**Parameters:**

`format` - the internal pixel format of the new Image2D
`width` - width of the created image in pixels; must be positive
`height` - height of the created image in pixels; must be positive
`image` - pixel data as a byte array, one byte per pixel
`palette` - palette entries as a byte array

**Throws:**

`java.lang.NullPointerException` - if `image` is null
`java.lang.NullPointerException` - if `palette` is null
`java.lang.IllegalArgumentException` - if `width <= 0`
`java.lang.IllegalArgumentException` - if `height <= 0`
`java.lang.IllegalArgumentException` - if `format` is not one of the symbolic constants listed above
`java.lang.IllegalArgumentException` - if `image.length < width*height`
`java.lang.IllegalArgumentException` - if `(palette.length < 256*C) && ((palette.length % C) != 0)`, where C is the number of color components (for instance, 3 for RGB)

## Image2D

```
public Image2D(int format,
               int width,
               int height)
```

Constructs an empty, mutable Image2D with the given dimensions. All pixels in the image are initialized to opaque white. The image contents can be set later by using the image as a rendering target in Graphics3D, or by using the `set` method. Note that only `RGB` and `RGBA` images can be used as rendering targets.

**Parameters:**

`format` - the internal pixel format of the new Image2D

`width` - width of the created image in pixels; must be positive
`height` - height of the created image in pixels; must be positive

**Throws:**

`java.lang.IllegalArgumentException` - if `width <= 0`
`java.lang.IllegalArgumentException` - if `height <= 0`
`java.lang.IllegalArgumentException` - if `format` is not one of the symbolic constants listed above

## Method Detail

### set

```
public void set(int x,
                int y,
                int width,
                int height,
                byte[] image)
```

Updates a rectangular area of this Image2D by copying pixels from a byte array. This method is only available for mutable images.

The area that is to be updated is specified in pixels, relative to the upper left corner of this Image2D. The area must lie completely within the bounds of the image.

Pixels in `image` are ordered from left to right and top to bottom. The number of bytes per pixel and the order of components are determined by the specified internal format; see the symbolic constants listed above.

**Parameters:**

`x` - the X coordinate of the area to update, relative to the top left corner
`y` - the Y coordinate of the area to update, relative to the top left corner
`width` - width of the area in the image to update, in pixels
`height` - height of the area in the image to update, in pixels
`image` - pixel data as a byte array

**Throws:**

`java.lang.NullPointerException` - if `image` is null
`java.lang.IllegalStateException` - if this Image2D is immutable
`java.lang.IllegalArgumentException` - if `x < 0`
`java.lang.IllegalArgumentException` - if `y < 0`
`java.lang.IllegalArgumentException` - if `width <= 0`
`java.lang.IllegalArgumentException` - if `height <= 0`
`java.lang.IllegalArgumentException` - if `(x + width) > getWidth`
`java.lang.IllegalArgumentException` - if `(y + height) > getHeight`
`java.lang.IllegalArgumentException` - if `image.length < width * height * bpp`, where `bpp` is the number of bytes per pixel (depends on the internal format specified at construction)

### isMutable

```
public boolean isMutable()
```

Queries whether this Image2D is mutable. The contents of a mutable image can be changed after construction, while the contents of an immutable image can not.

**Returns:**
*true* if this Image2D is mutable; *false* if it is immutable

## getFormat

```
public int getFormat()
```

Gets the internal format of this Image2D. Note that the format can not be changed after construction.

**Returns:**
the internal format of this Image2D; one of the symbolic constants listed above

## getWidth

```
public int getWidth()
```

Gets the width of this Image2D, in pixels. Note that the width and height can not be changed after construction.

**Returns:**
the width of this image

## getHeight

```
public int getHeight()
```

Gets the height of this Image2D, in pixels. Note that the width and height can not be changed after construction.

**Returns:**
the height of this image

javax.microedition.m3g

## Class IndexBuffer

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.IndexBuffer
```

**Direct Known Subclasses:**
TriangleStripArray

public abstract class **IndexBuffer**
extends Object3D

An abstract class defining how to connect vertices to form a geometric object.

Each IndexBuffer object defines a *submesh*, which is the basic rendering primitive in this API. In order to be rendered, a submesh must be associated with a VertexBuffer, defining the vertex attributes, and an Appearance, defining the surface attributes.

IndexBuffer is an abstract class that only provides functionality for querying the stored vertex indices. The actual index values as well as their interpretation are defined in each derived class. Currently, there is only one derived class, TriangleStripArray, which defines a submesh consisting of triangle strips. Other derived classes may be added in future revisions of this API, to support, for example, triangle lists, triangle fans, line strips and quad strips.

### Deferred exceptions

The indices in an IndexBuffer are only validated when rendering or picking. An exception is thrown by the `render` methods in Graphics3D and the `pick` methods in Group, if any of the indices are greater than or equal to the number of vertices in the associated VertexBuffer. The indices cannot be validated earlier, because they are allowed to be invalid at all other times except when they are used by the implementation, that is, when rendering or picking.

**See Also:**
Binary format

| **Method Summary** | |
|---|---|
| `int` | **`getIndexCount`**`()`<br>Returns the number of indices in this buffer. |
| `void` | **`getIndices`**`(int[] indices)`<br>Retrieves vertex indices for the rendering primitives stored in this buffer. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Method Detail

### getIndexCount

`public int getIndexCount()`

Returns the number of indices in this buffer. This many indices will be returned in a `getIndices` call. The number of indices returned depends on the type of low-level rendering primitives in the buffer: Currently, only triangles are supported, and there are three indices per triangle. Triangles are counted individually, disregarding triangle strips.

Note that implementations are allowed to optimize the index data internally. Different implementations may therefore report slightly different index counts for the same set of input primitives.

**Returns:**
the number of indices
**Since:**
M3G 1.1
**See Also:**
`getIndices`

### getIndices

`public void getIndices(int[] indices)`

Retrieves vertex indices for the rendering primitives stored in this buffer.

The indices returned describe the low-level rendering primitives in this buffer. The only such primitive currently supported is a triangle.

Triangles are returned individually, and each consecutive triplet of indices describes a single triangle. Triangle strips or implicit vs. explicit indices are not represented in the returned data. The triangles need not be in any particular order, but the order of indices in each triangle must preserve the winding of the triangle.

**Parameters:**
`indices` - array to store the returned vertex indices
**Throws:**
`java.lang.NullPointerException` - if `indices` is null
`java.lang.IllegalArgumentException` - if `indices.length < getIndexCount`
**Since:**
M3G 1.1

**javax.microedition.m3g**

# Class KeyframeSequence

```
java.lang.Object
  └─ javax.microedition.m3g.Object3D
       └─ javax.microedition.m3g.KeyframeSequence
```

public class **KeyframeSequence**
extends Object3D

Encapsulates animation data as a sequence of time-stamped, vector-valued keyframes. Each keyframe represents the value of an animated quantity at a specific time instant.

A KeyframeSequence can be associated with multiple *animation targets*, that is, animatable object properties, via multiple AnimationTrack objects. Available animation targets include node and texture transformations, Material parameters, Camera parameters, and so on. When applying an animation to its target, the actual value for the target is derived by interpolating the keyframe values in the associated KeyframeSequence object.

The number of vector components per keyframe is specified in the constructor and is the same for all keyframes in the sequence. The interpretation of the keyframes is determined by the animation target(s) the sequence is attached to. For example, 4-component keyframes are interpreted as quaternions when applied to the `ORIENTATION` target.

Five different functions are available for interpolating the keyframe values: `LINEAR` and `SPLINE`, their quaternion equivalents `SLERP` and `SQUAD`; and the simple `STEP` function. Each of these is described in the Field Summary. There are also two repeat modes, `LOOP` and `CONSTANT`, that affect the interpolation.

## Sequence time vs. world time

The internal *sequence time* t of a KeyframeSequence is derived from world time by its associated AnimationController (s). The formula for mapping world times to sequence times is given in the "Timing and speed control" section of the AnimationController class description. The sequence time is then used for interpolating between keyframe values as defined by the chosen interpolation function.

All (valid) keyframes in a KeyframeSequence must fall within a sequence time range of [0, D], where D is the duration of the keyframe sequence. The duration can be set with the `setDuration` method. For sequences using the `LOOP` repeat mode, the sequence time t is restricted into this range via a modulo operation, that is, by adding or subtracting a multiple of D such that $0 <= t < D$. For sequences using the `CONSTANT` mode, the value of t is unrestricted.

## Sequence repeat modes

The first valid keyframe in a `CONSTANT` sequence defines the interpolated value returned before this point in time. That is, with initial value $\mathbf{v}_0$ at time $t_0$, the interpolated value $\mathbf{v} = \mathbf{v}_0$ for values of time t such that $t < t_0$.

The final valid keyframe in a `CONSTANT` sequence defines the interpolated value returned after this point in time. That is, with final value $\mathbf{v}_{N-1}$ at time $t_{N-1}$, the interpolated value $\mathbf{v} = \mathbf{v}_{N-1}$ for values of time t such that $t >= t_{N-1}$.

A sequence using the `LOOP` repeat mode is interpolated as if the keyframes were replicated backward and forward indefinitely at a spacing equal to the given duration of the sequence. In this case, a keyframe which appears at time t will be treated as if it also appeared at time t + nD where n is any positive or negative integer and D is the duration of a single loop of the animation, given in `setDuration`. Note that this is *not* achieved by just the modulo operation on the sequence time described above.

In a looping sequence with N keyframes numbered [0, N-1], the successor of keyframe N-1 is keyframe 0, and the predecessor to keyframe 0 is keyframe N-1.

## Coincident keyframes

The specification allows several keyframes to coexist at the same position in time. This allows discontinuities in the animation sequences, which can be useful for example in incorporating cuts to camera animation. In the case of several keyframes coinciding, the one with the lowest index is always used for the final value of segments ending at that position in time; for segments starting at that position, the keyframe with the highest index is used for the starting value. For sequences in `LOOP` mode, the keyframes from the next or previous repeat of the sequence may also coincide with the keyframes of the current repeat if they are at the very end or very beginning of the sequence. The keyframes in the previous repeat are then treated as having lower indices, and the keyframes in the next repeat as having higher indices, than the keyframes in the current repeat.

Note that although any number of coincident keyframes can be specified, a maximum of four will ever be used in `SPLINE` or `SQUAD` interpolation; two in `LINEAR` or `SLERP` interpolation; and only the one with the highest index in `STEP` interpolation.

## Deferred exceptions

The validity of a keyframe sequence can be fully verified only when it is applied to an animation target, that is, in the `animate` method of Object3D. Any of the following conditions in a KeyframeSequence will then trigger an IllegalStateException:

- Duration of sequence not set;
- Duration less than the time of the last valid keyframe;
- Any keyframe times within the valid range in decreasing order.

## Implementation guidelines

Although independent of the keyframe values as such, the interpolation type and the repeat mode of a sequence are set here rather than in the AnimationTrack objects using the sequence. This is so that the implementation can sensibly cache spline tangents or other auxiliary data potentially required at runtime.

**See Also:**
Binary format, Example, `AnimationTrack`, `AnimationController`

| **Field Summary** | |
|---|---|
| `static int` | **`CONSTANT`**<br>A parameter to `setRepeatMode`, specifying that this sequence is to be played back just once and not repeated. |
| `static int` | **`LINEAR`**<br>A constructor parameter that specifies linear interpolation between keyframes. |

| | |
|---|---|
| static int | **LOOP**<br>A parameter to `setRepeatMode`, specifying that this sequence is to be repeated indefinitely. |
| static int | **SLERP**<br>A constructor parameter that specifies spherical linear interpolation of quaternions. |
| static int | **SPLINE**<br>A constructor parameter that specifies spline interpolation between keyframes. |
| static int | **SQUAD**<br>A constructor parameter that specifies spline interpolation of quaternions. |
| static int | **STEP**<br>A constructor parameter that specifies stepping from one keyframe value to the next. |

## Constructor Summary

**KeyframeSequence**`(int numKeyframes, int numComponents, int interpolation)`
Constructs a new keyframe sequence with specified interpolation method, number of components per keyframe, and number of keyframes.

## Method Summary

| | |
|---|---|
| int | **getComponentCount**`()`<br>Returns the number of components per keyframe in this sequence. |
| int | **getDuration**`()`<br>Gets the duration of this sequence. |
| int | **getInterpolationType**`()`<br>Returns the type of interpolation for this sequence. |
| int | **getKeyframe**`(int index, float[] value)`<br>Retrieves the time stamp and value of a single keyframe. |
| int | **getKeyframeCount**`()`<br>Returns the total number of keyframes in this sequence. |
| int | **getRepeatMode**`()`<br>Retrieves the current repeat mode of this KeyframeSequence. |
| int | **getValidRangeFirst**`()`<br>Returns the first keyframe of the current valid range for this sequence. |
| int | **getValidRangeLast**`()`<br>Returns the last keyframe of the current valid range for this sequence. |
| void | **setDuration**`(int duration)`<br>Sets the duration of this sequence in sequence time units. |
| void | **setKeyframe**`(int index, int time, float[] value)`<br>Sets the time position and value of the specified keyframe. |
| void | **setRepeatMode**`(int mode)`<br>Sets the repeat mode of this KeyframeSequence. |
| void | **setValidRange**`(int first, int last)`<br>Selects the range of keyframes that are included in the animation. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

### LINEAR

`public static final int LINEAR`

A constructor parameter that specifies linear interpolation between keyframes.

For a keyframe with value $\mathbf{v}_i$ at time $t_i$, where the following keyframe has a value $\mathbf{v}_{i+1}$ at time $t_{i+1}$, the interpolated value $\mathbf{v}$ is defined only for values of time t such that $t_i <= t < t_{i+1}$, as follows:

$$\mathbf{v} = (1-s)\mathbf{v}_i + s\mathbf{v}_{i+1}$$

where s is an interpolation factor in [0, 1) computed from the keyframe times:

$$s = (t - t_i) / (t_{i+1} - t_i)$$

**See Also:**
Constant Field Values

### SLERP

`public static final int SLERP`

A constructor parameter that specifies spherical linear interpolation of quaternions.

This type of interpolation will interpolate at constant speed along the shortest "great circle" path between two keyframe values along the surface of the hypersphere of unit quaternions.

Spherical linear interpolation between two keyframe values $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$ is defined as:

$$\mathrm{slerp}(s; \mathbf{q}_i, \mathbf{q}_{i+1}) = (\mathbf{q}_i \sin((1-s)a) + \mathbf{q}_{i+1} \sin(sa)) / \sin(a)$$

where a is the angle between the two quaternions and s is the interpolation factor defined for `LINEAR` interpolation.

Note that the shortest path between two quaternions is not the same as the shortest path between the corresponding 3D orientations. There are always two quaternions corresponding to a single 3D orientation, each of which denotes a different direction of interpolation along the great circle; thus, quaternions can encode up to 360 degrees of rotation between adjacent keyframes.

It is common practice in some applications to precondition quaternions prior to slerping so that the shorter interpolation path in 3D is always chosen. While this is useful in special cases, it does *not* yield the same result in general, and is therefore incompatible with more advanced features such as animation blending. Hence, implementations are explicitly disallowed from incorporating this practice, and *must* implement the general slerp routine instead. Authoring tools, however, are encouraged to present the preconditioning as an option when exporting keyframe data.

Also note that interpolation between diametrically opposed quaternions in successive keyframes is undefined. It is recommended that authoring tools should take steps to warn designers if this case is detected.

More details can be found in "Quaternion Algebra and Calculus" by David Eberly [see Related Literature].

**See Also:**

Constant Field Values

## SPLINE

`public static final int` **`SPLINE`**

A constructor parameter that specifies spline interpolation between keyframes. The keyframes will be interpolated with a Catmull-Rom spline adjusted to accommodate non-constant keyframe intervals.

For each curve segment i, we have the values $\mathbf{v}_i$ at time $t_i$, and $\mathbf{v}_{i+1}$ at time $t_{i+1}$. We also define tangents at the end points of the segment: $\mathbf{T}_i$ at the start point, and $\mathbf{T}_{i+1}$ at the end point.

Using the interpolation factor s defined for `LINEAR` interpolation, we can then express the interpolation of the curve as follows:

$$\mathbf{S} = \begin{vmatrix} s^3 \\ s^2 \\ s \\ 1 \end{vmatrix} \qquad \mathbf{H} = \begin{vmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{vmatrix} \qquad \mathbf{C} = \begin{vmatrix} \mathbf{v}_i \\ \mathbf{v}_{i+1} \\ \mathbf{T}^0{}_i \\ \mathbf{T}^1{}_{i+1} \end{vmatrix}$$

The value $\mathbf{v}_s$ of the curve at position s can be calculated using the formula:

$$\mathbf{v}_s = \mathbf{S}^T\mathbf{H}\mathbf{C}$$

The only thing left to define is the calculation of the tangent vectors $\mathbf{T}^{\{0,1\}}{}_i$. A standard Catmull-Rom spline assumes that the keyframe values are evenly spaced in time, and calculates the tangents as centered finite differences of the adjacent keyframes:

$$\mathbf{T}_i = (\mathbf{v}_{i+1} - \mathbf{v}_{i-1}) / 2$$

We apply additional scaling values to compensate for irregular keyframe timing, and the final tangents are:

$$\mathbf{T}^0_i = \mathbf{F}^-_i \mathbf{T}_i$$
$$\mathbf{T}^1_i = \mathbf{F}^+_i \mathbf{T}_i$$

where:

$$\mathbf{F}^-_i = 2\,(t_{i+1} - t_i) \,/\, (t_{i+1} - t_{i-1})$$
$$\mathbf{F}^+_i = 2\,(t_i - t_{i-1}) \,/\, (t_{i+1} - t_{i-1})$$
$\mathbf{F}^-_0 = \mathbf{F}^+_0 = \mathbf{F}^-_{N-1} = \mathbf{F}^+_{N-1} = 0,$     in a `CONSTANT`sequence

It is relatively easy to convert to this representation from piecewise Bezier splines (as used by 3ds max, for example) as long as the tangents are set up according to the above scheme. Conversion from other interpolating spline forms may not be exact, although any interpolating spline is guaranteed to pass through the keyframe values.

**See Also:**
Constant Field Values

## SQUAD

`public static final int` **`SQUAD`**

A constructor parameter that specifies spline interpolation of quaternions.

This interpolation method is similar to the `SPLINE` method, but using equivalent quaternion operations. The tangents for each keyframe are computed as centered finite differences, only this time via quaternion logarithms:

$$\mathbf{T}_n = (\ \log(\mathbf{q}_i^{-1}\mathbf{q}_{i+1}) + \log(\mathbf{q}_{i-1}^{-1}\mathbf{q}_i)\ ) \,/\, 2$$

Note that the operations above are *not* to be confused with scalar or vector operations. The notation $\mathbf{q}^{-1}$ denotes the inverse of quaternion $\mathbf{q}$; the multiplications are quaternion multiplications; and the logarithm is a quaternion logarithm, which essentially yields a 3-vector as a result.

Keyframe tangents are scaled to compensate for irregular keyframe timing as specified for `SPLINE` interpolation. This yields the "incoming" tangent $\mathbf{T}^0_i$ and the "outgoing" tangent $\mathbf{T}^1_i$. From the scaled tangents, intermediate quaternion values **a** and **b** are computed for use in interpolating the curve segments starting and ending at each keyframe:

$$\mathbf{a}_i = \mathbf{q}_i \exp(\ (\mathbf{T}^0_i - \log(\mathbf{q}_i^{-1}\mathbf{q}_{i+1})) \,/\, 2\ )$$
$$\mathbf{b}_i = \mathbf{q}_i \exp(\ (\log(\mathbf{q}_{i-1}^{-1}\mathbf{q}_i) - \mathbf{T}^1_i) \,/\, 2\ )$$

Finally, the interpolated value **q** at position s (as defined in `LINEAR`) for a curve segment between keyframes i and i + 1 is obtained by using `SLERP` interpolation, as follows:

$$\mathbf{q} = \mathrm{slerp}(\ 2s(1 - s);\ \mathrm{slerp}(s;\ \mathbf{q}_i,\ \mathbf{q}_{i+1}),\ \mathrm{slerp}(s;\ \mathbf{a}_i,\ \mathbf{b}_{i+1})\ )$$

For more information, refer to "Key Frame Interpolation via Splines and Quaternions" by David Eberly [see

Related Literature].

**See Also:**
Constant Field Values

### STEP

```
public static final int STEP
```

A constructor parameter that specifies stepping from one keyframe value to the next. The actual value of each keyframe is used, without further interpolation, from the time position of that keyframe until the time of the next keyframe.

For a keyframe with value **v** at time $t_i$, where the following keyframe is at time $t_{i+1}$, the value **v** is valid for all values of time t such that $t_i <= t < t_{i+1}$.

**See Also:**
Constant Field Values

### CONSTANT

```
public static final int CONSTANT
```

A parameter to `setRepeatMode`, specifying that this sequence is to be played back just once and not repeated.

**See Also:**
Constant Field Values

### LOOP

```
public static final int LOOP
```

A parameter to `setRepeatMode`, specifying that this sequence is to be repeated indefinitely.

**See Also:**
Constant Field Values

## Constructor Detail

### KeyframeSequence

```
public KeyframeSequence(int numKeyframes,
                        int numComponents,
                        int interpolation)
```

Constructs a new keyframe sequence with specified interpolation method, number of components per keyframe, and number of keyframes. All keyframes are initialized to the zero vector, with a time stamp of zero. The repeat mode is initially `CONSTANT` (not looping), with an undefined duration and the valid range spanning all keyframes.

A newly constructed sequence cannot be used in animation until the duration of the sequence has been set. The valid range, that is, the range of keyframes that are included in the animation, can be set with `setValidRange`. This may be desirable if keyframes are generated dynamically or streamed over the network.

The interpolation method is one of the symbolic constants defined above. The method must be compatible with the number of components in the keyframes. `STEP`, `LINEAR` and `SPLINE` can be specified for any type of keyframes. On the other hand, `SLERP` and `SQUAD` can only be specified for 4-component keyframes, which are then interpreted as quaternions.

**Parameters:**

`numKeyframes` - number of keyframes to allocate for this sequence
`numComponents` - number of components in each keyframe vector
`interpolation` - one of the interpolation modes listed above

**Throws:**

`java.lang.IllegalArgumentException` - if `numKeyframes < 1`
`java.lang.IllegalArgumentException` - if `numComponents < 1`
`java.lang.IllegalArgumentException` - if `interpolation` is not one of `LINEAR, SLERP, SPLINE, SQUAD, STEP`
`java.lang.IllegalArgumentException` - if `interpolation` is not a valid interpolation mode for keyframes of size `numComponents`

## Method Detail

### getComponentCount

```
public int getComponentCount()
```

Returns the number of components per keyframe in this sequence.

**Returns:**
the number of components

**Since:**
M3G 1.1

### getKeyframeCount

```
public int getKeyframeCount()
```

Returns the total number of keyframes in this sequence. Note that there may be fewer keyframes currently used for animation, controlled by the valid range.

**Returns:**
the number of keyframes

**Since:**
M3G 1.1

**See Also:**
`setValidRange`

## getInterpolationType

```
public int getInterpolationType()
```

Returns the type of interpolation for this sequence.

**Returns:**
the interpolation type; one of `LINEAR, SLERP, SPLINE, SQUAD, STEP`
**Since:**
M3G 1.1

## setKeyframe

```
public void setKeyframe(int index,
                        int time,
                        float[] value)
```

Sets the time position and value of the specified keyframe. The keyframe value is copied in from the given array. The length of the array must be at least equal to the size of a keyframe (`numComponents`). Refer to AnimationTrack documention for the order in which the keyframe components should be stored in the array for a particular target property.

If the interpolation type is `SLERP` or `SQUAD`, the keyframes are automatically normalized to yield unit quaternions for interpolation.

**Parameters:**
`index` - index of the keyframe to set
`time` - time position of the keyframe, in sequence time units
`value` - float array containing the keyframe value vector
**Throws:**
`java.lang.NullPointerException` - if `value` is null
`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getKeyframeCount)`
`java.lang.IllegalArgumentException` - if `value.length < getComponentCount`
`java.lang.IllegalArgumentException` - if `time < 0`
**See Also:**
`getKeyframe`

## getKeyframe

```
public int getKeyframe(int index,
                       float[] value)
```

Retrieves the time stamp and value of a single keyframe.

Note that if the interpolation type is SLERP or SQUAD, the keyframes are automatically normalized upon setting. The values returned here may therefore be different from the original input values.

**Parameters:**
`index` - index of the keyframe to retrieve
`value` - float array to store the keyframe value vector, or null to only return the time stamp

**Returns:**
the time value of the keyframe

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getKeyframeCount)`
`java.lang.IllegalArgumentException` - if `(value != null) && (value.length < getComponentCount)`

**Since:**
M3G 1.1

**See Also:**
`setKeyframe`

## setValidRange

```
public void setValidRange(int first,
                          int last)
```

Selects the range of keyframes that are included in the animation. Keyframes outside of that range are ignored by the `animate` method in Object3D.

Setting the valid range shorter than the whole sequence enables the application to use the sequence as a circular buffer when generating new keyframe data on the fly, for example. In a typical case, however, the valid range would span the whole sequence.

The valid keyframe range is always interpreted in the direction of ascending indices. If `first <= last`, the valid keyframes are those at the indices:

```
first, first+1, ..., last
```

If `last < first`, the valid range wraps around and the valid keyframe indices are:

```
first, first+1, ..., getKeyframeCount()-1, 0, 1, ..., last
```

The time position of each keyframe in the active range must be greater than or equal to that of the preceding keyframe; if this is not the case, `Object3D.animate` will throw an exception. The time stamps must be in non-decreasing order, because otherwise the interpolated values between keyframes would be undefined. Note that having two or more keyframes with the same time stamp is specifically allowed.

**Parameters:**
`first` - index of the first valid keyframe
`last` - index of the last valid keyframe

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `(first < 0) || (first >= getKeyframeCount)`
`java.lang.IndexOutOfBoundsException` - if `(last < 0) || (last >= getKeyframeCount)`

**See Also:**
`getValidRangeFirst`, `getValidRangeLast`

## getValidRangeFirst

`public int getValidRangeFirst()`

Returns the first keyframe of the current valid range for this sequence.

**Returns:**
the index of the first valid keyframe
**Since:**
M3G 1.1
**See Also:**
`setValidRange`

## getValidRangeLast

`public int getValidRangeLast()`

Returns the last keyframe of the current valid range for this sequence.

**Returns:**
the index of the last valid keyframe
**Since:**
M3G 1.1
**See Also:**
`setValidRange`

## setDuration

`public void setDuration(int duration)`

Sets the duration of this sequence in sequence time units. The duration of a keyframe sequence, as used in animation playback, is determined by the value set here, irrespective of the time stamps of individual keyframes, and irrespective of which keyframes happen to be in the valid range at any given time.

The duration D is also used when interpolating looping keyframe sequences. The time interval from the last valid keyframe to the first valid keyframe of the next cycle is calculated as follows:

$$D - (t_{last} - t_{first})$$

where $t_{first}$ and $t_{last}$ are the time stamps of the first and last keyframe, respectively, in the valid range (see `setValidRange`). Note that they are not necessarily the first and last keyframe of the whole sequence.

The duration of the sequence must not be less than the time stamp of the last valid keyframe ($t_{last}$), as otherwise the above formula would yield a negative time interval. Since the duration and the valid range can both be changed at any time, this condition is only enforced by the `animate` method in Object3D.

**Parameters:**
`duration` - duration of the valid range of the sequence

**Throws:**

`java.lang.IllegalArgumentException` - if `duration <= 0`

**See Also:**

`getDuration`

## getDuration

```
public int getDuration()
```

Gets the duration of this sequence.

**Returns:**

the duration of this sequence in sequence time units

**See Also:**

`setDuration`

## setRepeatMode

```
public void setRepeatMode(int mode)
```

Sets the repeat mode of this KeyframeSequence. There are two alternatives, `LOOP` and `CONSTANT`.

A looping sequence always loops back to the beginning from the end and has an interpolated segment from the last valid keyframe to the first.

A constant sequence maintains the first valid keyframe value from the beginning of the sequence to the actual time of that keyframe, and the last valid keyframe value from that keyframe to the end time of the sequence and beyond.

**Parameters:**

`mode` - the repeat mode to set

**Throws:**

`java.lang.IllegalArgumentException` - if `mode` is not one of `CONSTANT, LOOP`

**See Also:**

`getRepeatMode`

## getRepeatMode

```
public int getRepeatMode()
```

Retrieves the current repeat mode of this KeyframeSequence.

**Returns:**

the current repeat mode; `CONSTANT` or `LOOP`

**See Also:**

`setRepeatMode`

**javax.microedition.m3g**

# Class Light

```
java.lang.Object
  └─javax.microedition.m3g.Object3D
      └─javax.microedition.m3g.Transformable
          └─javax.microedition.m3g.Node
              └─javax.microedition.m3g.Light
```

public class **Light**
extends Node

A scene graph node that represents different kinds of light sources.

Light sources are used to determine the color of each object according to its Material attributes, as described in more detail in the Material class documentation.

## Light source types

Four types of light sources are available. In the order of increasing computational complexity, these are the *ambient light*, *directional light*, *omnidirectional light* and *spot light*. Their characteristics are described below.

- An **ambient** light source illuminates all objects in the scene from all directions. The intensity of light coming from an ambient light source is the same everywhere in the scene. The position and direction of an ambient light source therefore have no effect.

- A **directional** light source corresponds to sunlight in the real world. It illuminates all objects in the scene from the same direction, and with a constant intensity. Similar to ambient light, the position of a directional light source has no effect. The direction of the light is along the negative Z axis of the Light node's local coordinate system.

- An **omnidirectional** light source, also known as a *point light*, casts equal illumination in all directions from the origin of the Light node's local coordinate system. The intensity of light coming from an omnidirectional light source can be attenuated with distance. The orientation of an omnidirectional Light node has no effect; only the position matters.

- A **spot** light source casts a cone of light centered around the direction of its negative Z axis. The concentration of light within the cone can be adjusted with the spot exponent. The intensity of light coming from a spot light can be attenuated with distance from the source. Both the orientation and the position of the Light node have an effect with spot lights.

The type of a light source can be changed at any time. This is useful for switching to a simpler lighting model when the distance to an object increases over a certain threshold, for example.

## Light color and intensity

The RGB intensity contributed to the lighting calculation by a Light is (IR, IG, IB), where I is the intensity of the Light and (R, G, B) is its color. Note that while 1.0 is a nominal full intensity, applications may use values higher than that for more control over highlights, for example. The intensity may also be set to negative to specify an "antilight" or "dark".

In the case of an ambient light source, the final RGB intensity represents the ambient color component only; the diffuse and specular components are implicitly set to zero. In the case of a directional, omni or spot light, the final intensity represents both the diffuse and specular components, while the ambient component is correspondingly set to zero.

## Light source selection

The set of meshes affected by a Light can be limited using the scope of the Light and each Mesh. A Light node is only included in the lighting calculations for a mesh if the scope of the mesh matches the scope of the Light. See the Node class description for more information.

Lights can be turned on and off using `Node.setRenderingEnable(boolean)`. The corresponding picking enable flag has no effect, because Lights are always ignored when picking.

## Implementation guidelines

The number of Lights matching with a single Mesh may be greater than the maximum number of concurrent lights that the implementation can support (N). In this case, the implementation may choose any N lights, as long as the selection is deterministic. For best results, the implementation may use a suitable heuristic to select the N lights that have the most effect on the rendered appearance of the mesh. The light selection may even be done separately for each submesh. The maximum number of concurrent lights can be queried from `getProperties`.

**See Also:**
Binary format, `Material`

| **Field Summary** | |
|---|---|
| `static int` | **`AMBIENT`**<br>A parameter to `setMode`, specifying an ambient light source. |
| `static int` | **`DIRECTIONAL`**<br>A parameter to `setMode`, specifying a directional light source. |
| `static int` | **`OMNI`**<br>A parameter to `setMode`, specifying an omnidirectional light source. |
| `static int` | **`SPOT`**<br>A parameter to `setMode`, specifying a spot light source. |

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

| **Constructor Summary** |
|---|
| **`Light`**`()`<br>Constructs a new Light with default values. |

| Method Summary | |
|---|---|
| `int` | **`getColor`**`()`<br>Retrieves the current color of this Light. |
| `float` | **`getConstantAttenuation`**`()`<br>Retrieves the current constant attenuation coefficient for this Light. |
| `float` | **`getIntensity`**`()`<br>Retrieves the current intensity of this Light. |
| `float` | **`getLinearAttenuation`**`()`<br>Retrieves the current linear attenuation coefficient for this Light. |
| `int` | **`getMode`**`()`<br>Retrieves the current type of this Light. |
| `float` | **`getQuadraticAttenuation`**`()`<br>Retrieves the current quadratic attenuation coefficient for this Light. |
| `float` | **`getSpotAngle`**`()`<br>Retrieves the current spot angle of this Light. |
| `float` | **`getSpotExponent`**`()`<br>Retrieves the current spot exponent for this Light. |
| `void` | **`setAttenuation`**`(float constant, float linear, float quadratic)`<br>Sets the attenuation coefficients for this Light. |
| `void` | **`setColor`**`(int RGB)`<br>Sets the color of this Light. |
| `void` | **`setIntensity`**`(float intensity)`<br>Sets the intensity of this Light. |
| `void` | **`setMode`**`(int mode)`<br>Sets the type of this Light. |
| `void` | **`setSpotAngle`**`(float angle)`<br>Sets the spot cone angle for this Light. |
| `void` | **`setSpotExponent`**`(float exponent)`<br>Sets the spot exponent for this Light. |

| Methods inherited from class javax.microedition.m3g.Node |
|---|
| `align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope` |

| Methods inherited from class javax.microedition.m3g.Transformable |
|---|
| `getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate` |

**Methods inherited from class javax.microedition.m3g.Object3D**

`addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject`

## Field Detail

### AMBIENT

`public static final int AMBIENT`

A parameter to `setMode`, specifying an ambient light source.

**See Also:**
Constant Field Values

### DIRECTIONAL

`public static final int DIRECTIONAL`

A parameter to `setMode`, specifying a directional light source.

**See Also:**
Constant Field Values

### OMNI

`public static final int OMNI`

A parameter to `setMode`, specifying an omnidirectional light source.

**See Also:**
Constant Field Values

### SPOT

`public static final int SPOT`

A parameter to `setMode`, specifying a spot light source.

**See Also:**
Constant Field Values

## Constructor Detail

### Light

`public Light()`

Constructs a new Light with default values. The default values are as follows:

- mode: `DIRECTIONAL`
- color : 0x00FFFFFF (1.0, 1.0, 1.0)
- intensity : 1.0
- attenuation : (1, 0, 0)
- spot angle : 45 degrees
- spot exponent : 0.0

## Method Detail

### setMode

`public void setMode(int mode)`

Sets the type of this Light. See the class description for more information.

**Parameters:**
`mode` - the mode to set; one of the symbolic constants listed above

**Throws:**
`java.lang.IllegalArgumentException` - if `mode` is not one of `AMBIENT, DIRECTIONAL, OMNI, SPOT`

**See Also:**
`getMode`

### getMode

`public int getMode()`

Retrieves the current type of this Light.

**Returns:**
the current mode of this Light; one of the symbolic constants listed above

**See Also:**
`setMode`

### setIntensity

`public void setIntensity(float intensity)`

Sets the intensity of this Light. The RGB color of this Light is multiplied component-wise with the intensity before computing the lighting equation. See the class description for more information.

**Parameters:**
`intensity` - the intensity to set; may be negative or zero

**See Also:**
`getIntensity`

## getIntensity

```
public float getIntensity()
```

Retrieves the current intensity of this Light.

**Returns:**
the current intensity of this Light

**See Also:**
`setIntensity`

## setColor

```
public void setColor(int RGB)
```

Sets the color of this Light. Depending on the type of light, this represents either the ambient color or both the diffuse and specular colors. See the class description for more information. The high order byte of the color value (that is, the alpha component) is ignored.

**Parameters:**
`RGB` - the color to set for this Light in 0x00RRGGBB format

**See Also:**
`getColor`

## getColor

```
public int getColor()
```

Retrieves the current color of this Light. The high order byte of the color value (that is, the alpha component) is guaranteed to be zero.

**Returns:**
the current color of this Light in 0x00RRGGBB format

**See Also:**
`setColor`

## setSpotAngle

```
public void setSpotAngle(float angle)
```

Sets the spot cone angle for this Light. The effect of this Light is restricted to a cone of `angle` degrees around the negative Z axis of the Light.

Note that this setting has no effect unless the type of this Light is (or is later set to) `SPOT`.

**Parameters:**
`angle` - the spot angle to set, in degrees
**Throws:**
`java.lang.IllegalArgumentException` - if `angle` is not in [0, 90]
**See Also:**
`getSpotAngle`

## getSpotAngle

```
public float getSpotAngle()
```

Retrieves the current spot angle of this Light.

**Returns:**
the current spot angle of this Light
**See Also:**
`setSpotAngle`

## setSpotExponent

```
public void setSpotExponent(float exponent)
```

Sets the spot exponent for this Light. The spot exponent controls the distribution of the intensity of this Light within the spot cone, such that larger values yield a more concentrated cone. The default spot exponent is 0.0, resulting in a uniform light distribution.

Note that this setting has no effect unless the type of this Light is (or is later set to) `SPOT`.

**Parameters:**
`exponent` - the spot light exponent to set
**Throws:**
`java.lang.IllegalArgumentException` - if `exponent` is not in [0, 128]
**See Also:**
`getSpotExponent`

## getSpotExponent

```
public float getSpotExponent()
```

Retrieves the current spot exponent for this Light.

**Returns:**
the current spot exponent of this Light
**See Also:**
`setSpotExponent`

### setAttenuation

```
public void setAttenuation(float constant,
                           float linear,
                           float quadratic)
```

Sets the attenuation coefficients for this Light. The attenuation factor is

$$1 / (c + l\,d + q\,d^2)$$

where d is the distance between the light and the vertex being lighted, and c, l, q are the constant, linear, and quadratic coefficients. The default attenuation coefficients are (1, 0, 0), resulting in no attenuation.

Note that this setting has no effect unless the type of this Light is (or is later set to) `OMNI` or `SPOT`.

**Parameters:**

`constant` - the constant attenuation coefficient to set
`linear` - the linear attenuation coefficient to set
`quadratic` - the quadratic attenuation coefficient to set

**Throws:**

`java.lang.IllegalArgumentException` - if any of the parameter values are negative
`java.lang.IllegalArgumentException` - if all of the parameter values are zero

### getConstantAttenuation

```
public float getConstantAttenuation()
```

Retrieves the current constant attenuation coefficient for this Light.

**Returns:**

the current constant attenuation coefficient

**See Also:**

`setAttenuation`

### getLinearAttenuation

```
public float getLinearAttenuation()
```

Retrieves the current linear attenuation coefficient for this Light.

**Returns:**

the current linear attenuation coefficient

**See Also:**

`setAttenuation`

### getQuadraticAttenuation

```
public float getQuadraticAttenuation()
```

Retrieves the current quadratic attenuation coefficient for this Light.

**Returns:**

the current quadratic attenuation coefficient

**See Also:**

`setAttenuation`

**javax.microedition.m3g**

# Class Loader

```
java.lang.Object
  └─ javax.microedition.m3g.Loader
```

public class **Loader**
extends java.lang.Object

Downloads and deserializes scene graph nodes and node components, as well as entire scene graphs. Downloading ready-made pieces of 3D content from an M3G file is generally the most convenient way for an application to create and populate a 3D scene.

## Supported data types

The Loader can deserialize instances of any class derived from Object3D. These include scene graph nodes such as World, Group, Camera and Light; attribute classes such as Material, Appearance and Texture2D; animation classes such as AnimationTrack and KeyframeSequence; and so on. No other types of objects are supported.

The data to be loaded must constitute a valid M3G file. Alternatively, it may be a PNG image file, in which case a single, immutable Image2D object is returned, with the pixel format of the Image2D corresponding to the color type of the PNG. Some implementations may support other formats as well. If the data is not in a supported format, is otherwise invalid, or can not be loaded for some other reason, an exception is thrown.

## Using the Loader

The Loader class cannot be instantiated, and its only members are the two static `load` methods. The methods are otherwise identical, but one of them takes in a byte array, while the other takes a named resource, such as a URI or an individual file in the JAR package. Named resources must always have an absolute path, otherwise the results are undefined. For example, loading "foobar.m3g" produces undefined results, whereas loading "/foobar.m3g" is well-defined. Furthermore, named resources are treated as case-sensitive. For example, "foobar.m3g" is not the same file as "foobar.M3G".

Any external references in the given file or byte array are followed recursively. When using the `load` variant that takes in a URI, the references may be absolute or relative, but when using the byte array variant, only absolute references are allowed. External references are also treated as case-sensitive.

The `load` methods only return once the entire contents of the given file (or byte array) have been loaded, including any referenced files. This means that displaying content while downloading (progressive loading) is not supported.

## Managing the loaded objects

The `load` methods return an array of Object3Ds. These are the root level objects in the file; in other words, those objects that are not referenced by any other objects. The array is guaranteed not to contain any null objects, but the order of the objects in the array is undefined.

The non-root objects (often the majority) can be found by following references recursively, starting from the root objects. This can be done conveniently with the `getReferences` method in Object3D. Another way to find a specific object is to tag it with a known user ID at the authoring stage, and search for that among the loaded objects using the `find` method.. See the class description for Object3D for more information.

Since the root-level objects are returned in an Object3D array, the application must find out their concrete types before using their full functionality. In the typical case, when the content is developed in conjunction with the application code and deployed in the same JAR file, the application should know what the root-level objects are. If this information is not available, or there is a need to check that the objects are as expected, the application can use the run-time type information that is built into Java. For example, a simple animation player application might want to check that the downloaded object is indeed a World, and display an appropriate error message otherwise.

## Validity of the loaded objects

The set of objects returned by the Loader, comprising the root level objects as well as their descendants, is guaranteed to be valid and consistent with the API. In other words, it should be possible to construct the same scene graph using the API with no exceptions being thrown. For example, if a Mesh object is returned, the application can rest assured that it is in a state that can be reached via the API.

However, it is not guaranteed that the loaded content is renderable. Conditions that cause deferred exceptions are not checked for, and may exist within the loaded objects. This allows fragmentary scene graphs, which are invalid for rendering, to be loaded and assembled by the application into a valid scene graph.

## Implementation guidelines

Implementations must not rely on the file extension (such as ".png") to determine the type of the file that is to be loaded. Instead, if the MIME type is available, that should be used to screen out unrecognized files without examining the contents. If the MIME type is not available (such as when loading from a byte array), or it does indicate a supported format, the implementation must ascertain the file type based on its contents. M3G files can be recognized from the file identifier, and PNG files from their corresponding signature.

Implementations must conform to the requirements set forth in the PNG specification, section Conformance of PNG decoders. In addition, the tRNS chunk (transparency information) must be fully supported. The implementation may also support other ancillary chunks. The pixel format of the resulting Image2D must be determined from the color type and transparency information contained in the PNG file, as specified in the table below.

| PNG color type | Image2D pixel format |
|---|---|
| Greyscale (type 0) | `LUMINANCE` |
| Greyscale (type 0) + tRNS | `LUMINANCE_ALPHA` |
| Truecolor (type 2) | `RGB` |
| Truecolor (type 2) + tRNS | `RGBA` |
| Indexed-color (type 3) | `RGB` |
| Indexed-color (type 3) + tRNS | `RGBA` |
| Greyscale with alpha (type 4) | `LUMINANCE_ALPHA` |
| Truecolor with alpha (type 6) | `RGBA` |

**See Also:**
M3G (JSR-184) file format, PNG file format

**Example:**
**A code fragment illustrating the use of Loader and `find`.**

```
Object3D[] roots=null;

try {
    // Load a World and an individual Mesh over http.

    roots = Loader.load("http://www.example.com/myscene.m3g");
} catch(IOException e) {
    // couldn't open the connection, or invalid data in the file
}

// The root objects must be cast from Object3D to their immediate types
// (Mesh and World) before their full functionality can be used. Since
// the relative ordering of the root objects is unspecified, we can't
// assume that the World object is always at index 0, for example. Instead,
// we identify the World by its user ID of 1, which we have assigned to it
// at the authoring stage.

World myWorld;                      // contains our entire scene graph
Mesh myMesh;                        // an individual mesh for immediate mode

if (roots[0].getUserID() == 1) {    // our World has a user ID of 1
    myWorld = (World) roots[0];
    myMesh = (Mesh) roots[1];
} else {
    myWorld = (World) roots[1];
    myMesh = (Mesh) roots[0];
}

// Turn on perspective correction for the Mesh.

Appearance a = myMesh.getAppearance(0);  // get the appearance of the mesh
PolygonMode p = a.getPolygonMode();      // get its polygon attributes
p.setPerspectiveCorrectionEnable(true);  // enable perspective correction

// Find a specific Camera node in the World, and set it as the currently
// active camera in the world. We've previously assigned the userID "10"
// to that camera node.

Camera myCamera = (Camera) myWorld.find(10);
myWorld.setActiveCamera(myCamera);

// Load an individual PNG file.

Image2D textureImage=null;

try {
     textureImage = (Image2D)Loader.load("/texture.png")[0];
} catch(IOException e) {
     // couldn't load the PNG file
}
```

## Method Summary

| | |
|---|---|
| `static Object3D[]` | **`load`**`(byte[] data, int offset)`<br>Deserializes Object3D instances from the given byte array, starting at the given offset. |

| | |
|---|---|
| `static Object3D[]` | `load(java.lang.String name)` Deserializes Object3D instances from the named resource. |

## Method Detail

### load

```
public static Object3D[] load(java.lang.String name)
                       throws java.io.IOException
```

Deserializes Object3D instances from the named resource. The name of the resource is as defined by `Class.getResourceAsStream(name)`, or a URI. The types of data that can be loaded with this method are defined in the class description.

**Parameters:**
`name` - name of the resource to load from

**Returns:**
an array of newly created Object3D instances

**Throws:**
`java.lang.NullPointerException` - if `name` is null
`java.io.IOException` - if `name`, or any resource referenced from it, cannot be resolved or accessed
`java.io.IOException` - if the data in `name`, or in any resource referenced from it, is not in accordance with the M3G and PNG file format specifications
`java.lang.SecurityException` - if the application does not have the security permission to open a connection to load the data

### load

```
public static Object3D[] load(byte[] data,
                              int offset)
                       throws java.io.IOException
```

Deserializes Object3D instances from the given byte array, starting at the given offset. The types of data that can be loaded with this method are defined in the class description. The byte array must not contain any relative references (such as "/pics/texture.png"), but complete URIs are allowed.

**Parameters:**
`data` - byte array containing the serialized objects to load
`offset` - index at which to start reading the `data` array

**Returns:**
an array of newly created Object3D instances

**Throws:**
`java.lang.NullPointerException` - if `data` is null
`java.lang.IndexOutOfBoundsException` - if (`offset` < 0) || (`offset` >= `data.length`)
`java.io.IOException` - if any external references in `data` cannot be resolved or accessed
`java.io.IOException` - if the data in `data`, or in any resource referenced from it, is not in accordance with the M3G and PNG file format specifications

`java.lang.SecurityException` - if the application does not have the security permission to open a connection to load an external reference

**javax.microedition.m3g**

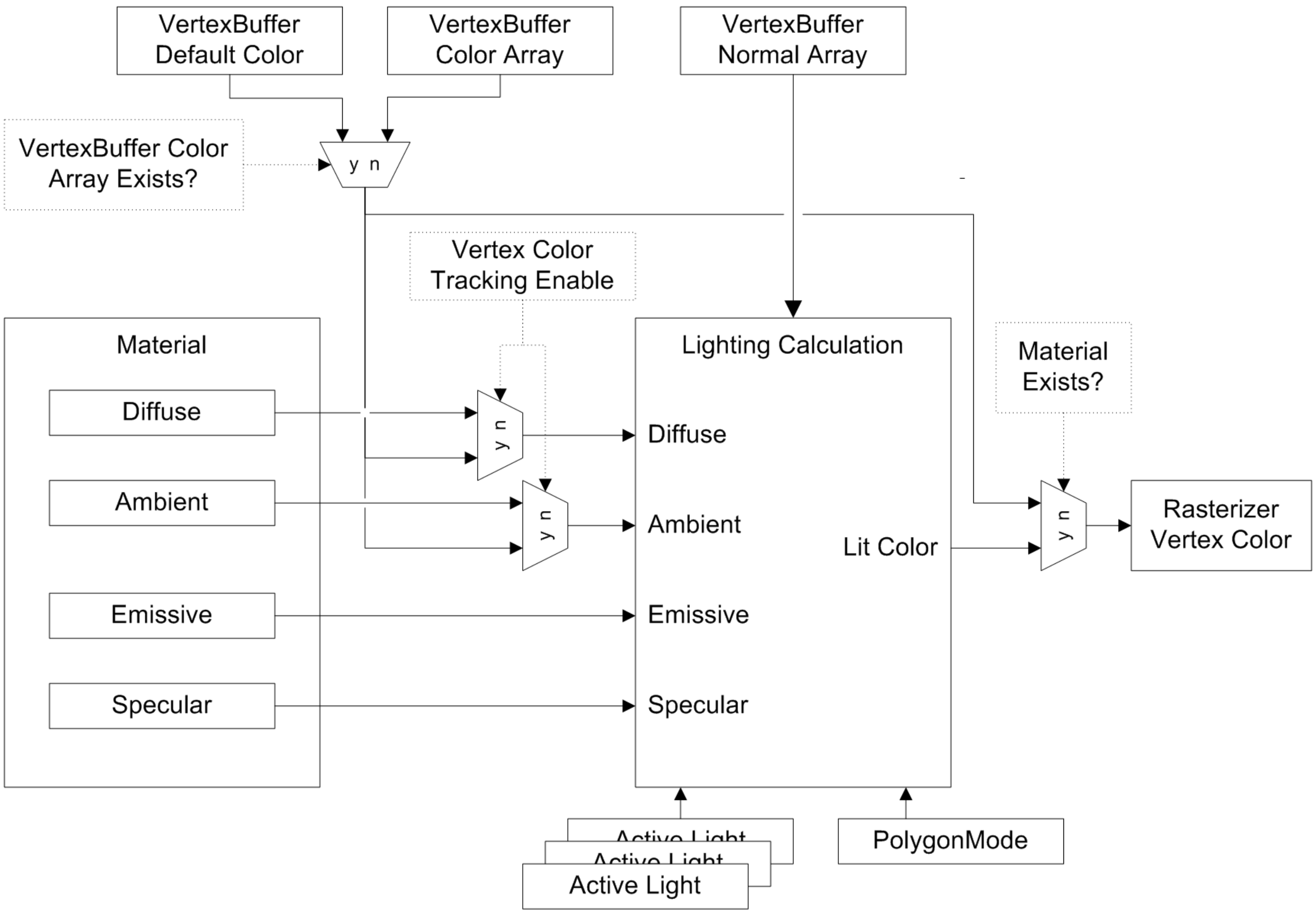
## Class Material

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Material
```

public class **Material**
extends Object3D

An Appearance component encapsulating material attributes for lighting computations. Other attributes required for lighting are defined in Light, PolygonMode and VertexBuffer.

The diagram below illustrates how the final, *lit* color is obtained for a vertex. Lighting is disabled for a submesh if it has a null Material, and enabled otherwise. If lighting is disabled, the final vertex color is taken from the associated VertexBuffer as such. If lighting is enabled, the final color is computed according to the OpenGL 1.3 lighting equation (p. 48), using the material colors specified here. Finally, if vertex color tracking is enabled, the `AMBIENT` and `DIFFUSE` material colors are replaced with the per-vertex colors or the default color obtained from the VertexBuffer.



## Implementation guidelines

Lighting is computed according to the OpenGL 1.3 specification, section 2.13.1, with the following exceptions:

- the secondary color is not supported;
- the same Material is used for both the front face and the back face;
- vertex color tracking is limited to `AMBIENT_AND_DIFFUSE`;
- for an ambient Light, the diffuse and specular intensities are zero;
- for a directional or positional Light, the ambient intensity is zero;
- the diffuse and specular Light intensities can not be set separately;
- the global scene ambient color $\mathbf{a}_{cs}$ is not supported;

**See Also:**
Binary format

## Field Summary

| | |
|---|---|
| `static int` | **`AMBIENT`**<br>A parameter to `setColor` and `getColor`, specifying that the ambient color component is to be set or retrieved. |
| `static int` | **`DIFFUSE`**<br>A parameter to `setColor` and `getColor`, specifying that the diffuse color component is to be set or retrieved. |
| `static int` | **`EMISSIVE`**<br>A parameter to `setColor` and `getColor`, specifying that the emissive color component is to be set or retrieved. |
| `static int` | **`SPECULAR`**<br>A parameter to `setColor` and `getColor`, specifying that the specular color component is to be set or retrieved. |

## Constructor Summary

| |
|---|
| **`Material`**`()`<br>Creates a Material object with default values. |

## Method Summary

| | |
|---|---|
| `int` | **`getColor`**`(int target)`<br>Gets the value of the specified color component of this Material. |
| `float` | **`getShininess`**`()`<br>Gets the current shininess of this Material. |
| `boolean` | **`isVertexColorTrackingEnabled`**`()`<br>Queries whether vertex color tracking is enabled. |
| `void` | **`setColor`**`(int target, int ARGB)`<br>Sets the given value to the specified color component(s) of this Material. |

| | |
|---|---|
| void | `setShininess(float shininess)` Sets the shininess of this Material. |
| void | `setVertexColorTrackingEnable(boolean enable)` Enables or disables vertex color tracking. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

### AMBIENT

`public static final int AMBIENT`

A parameter to `setColor` and `getColor`, specifying that the ambient color component is to be set or retrieved.

**See Also:**
Constant Field Values

### DIFFUSE

`public static final int DIFFUSE`

A parameter to `setColor` and `getColor`, specifying that the diffuse color component is to be set or retrieved.

**See Also:**
Constant Field Values

### EMISSIVE

`public static final int EMISSIVE`

A parameter to `setColor` and `getColor`, specifying that the emissive color component is to be set or retrieved.

**See Also:**
Constant Field Values

### SPECULAR

`public static final int SPECULAR`

A parameter to `setColor` and `getColor`, specifying that the specular color component is to be set or retrieved.

**See Also:**
Constant Field Values

## Constructor Detail

### Material

```
public Material()
```

Creates a Material object with default values. The default values are:

- vertex color tracking : *false* (disabled)
- ambient color : 0x00333333 (0.2, 0.2, 0.2, 0.0)
- diffuse color : 0xFFCCCCCC (0.8, 0.8, 0.8, 1.0)
- emissive color : 0x00000000 (0.0, 0.0, 0.0, 0.0)
- specular color : 0x00000000 (0.0, 0.0, 0.0, 0.0)
- shininess : 0.0

Note that even though the alpha component can be set for all color components, it is ignored for all but the diffuse component.

## Method Detail

### setColor

```
public void setColor(int target,
                     int ARGB)
```

Sets the given value to the specified color component(s) of this Material. The color components to set are specified as an inclusive OR of one or more of the symbolic constants listed above. The color is given in ARGB format, but the alpha component is ignored for all but the diffuse color.

**Parameters:**
`target` - a bitmask of color component identifiers
`ARGB` - color for the target property (or properties) in 0xAARRGGBB format
**Throws:**
`java.lang.IllegalArgumentException` - if `target` has a value other than an inclusive OR of one or more of `AMBIENT, DIFFUSE, EMISSIVE, SPECULAR`
**See Also:**
`getColor`

### getColor

`public int getColor(int target)`

Gets the value of the specified color component of this Material. The alpha component of the returned value is guaranteed to be zero for all but the diffuse color component.

**Parameters:**
`target` - exactly one of `AMBIENT, DIFFUSE, EMISSIVE, SPECULAR`
**Returns:**
the current color of the target property in 0xAARRGGBB format
**Throws:**
`java.lang.IllegalArgumentException` - if `target` has a value other than one of those listed above
**See Also:**
`setColor`

### setShininess

`public void setShininess(float shininess)`

Sets the shininess of this Material. Shininess is the specular exponent term in the lighting equation, and it can take on values between [0, 128]. Large values of shininess make the specular highlights more concentrated, and small values make them more spread out.

**Parameters:**
`shininess` - the specular exponent value to set for this Material
**Throws:**
`java.lang.IllegalArgumentException` - if `shininess` is not in [0, 128]
**See Also:**
`getShininess`

### getShininess

`public float getShininess()`

Gets the current shininess of this Material.

**Returns:**
the current specular exponent value of this Material
**See Also:**
`setShininess`

### setVertexColorTrackingEnable

`public void setVertexColorTrackingEnable(boolean enable)`

Enables or disables vertex color tracking. When enabled, the `AMBIENT` and `DIFFUSE` material colors will take on color values from the associated VertexBuffer on a per-vertex basis. The ambient and diffuse color values of this Material are ignored in that case.

**Parameters:**

`enable` - *true* to turn vertex color tracking on; *false* to turn it off

## isVertexColorTrackingEnabled

```
public boolean isVertexColorTrackingEnabled()
```

Queries whether vertex color tracking is enabled.

**Returns:**
*true* if vertex color tracking is enabled; *false* if it's disabled

**javax.microedition.m3g**
# Class Mesh

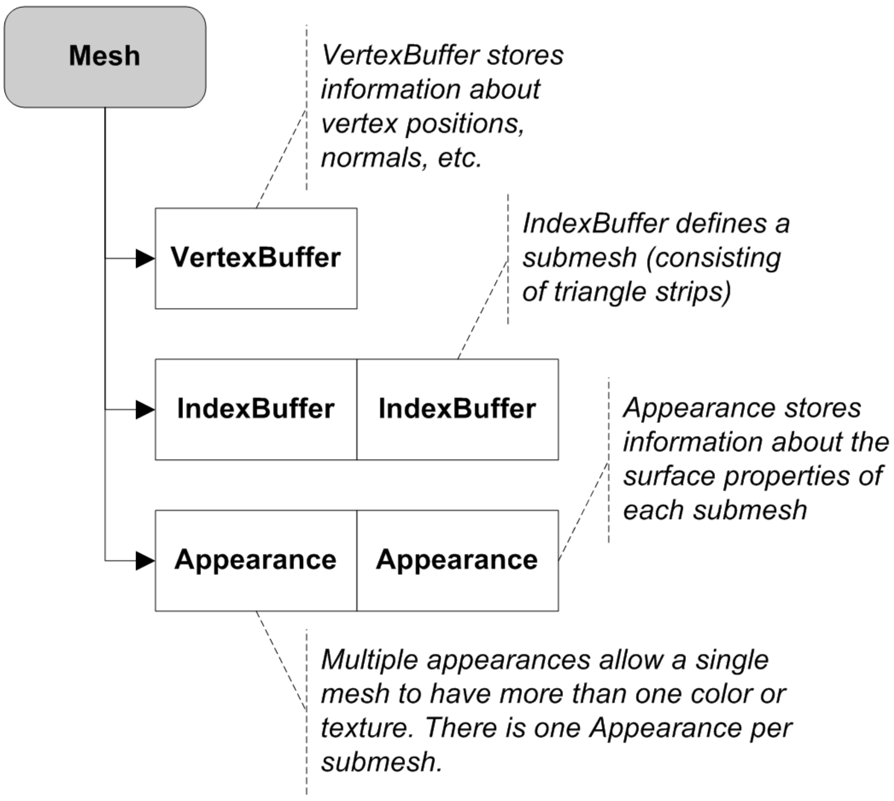
```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
          └ javax.microedition.m3g.Node
              └ javax.microedition.m3g.Mesh
```

**Direct Known Subclasses:**
MorphingMesh, SkinnedMesh

public class **Mesh**
extends Node

A scene graph node that represents a 3D object defined as a polygonal surface.

This class represents a conventional rigid body mesh, while the derived classes MorphingMesh and SkinnedMesh extend it with capabilities to transform vertices independently of each other. The structure of a basic Mesh is shown in the figure below.


Mesh
VertexBuffer stores information about vertex positions, normals, etc.
VertexBuffer
IndexBuffer defines a submesh (consisting of triangle strips)
IndexBuffer
IndexBuffer
Appearance stores information about the surface properties of each submesh
Appearance
Appearance
Multiple appearances allow a single mesh to have more than one color or texture. There is one Appearance per submesh.

A Mesh is composed of one or more submeshes and their associated Appearances. A submesh is an array of triangle strips defined by an IndexBuffer object. The triangle strips are formed by indexing the vertex coordinates and other vertex attributes in an associated VertexBuffer. All submeshes in a Mesh share the same VertexBuffer. However, in the case of a MorphingMesh, a weighted linear combination of multiple VertexBuffers is used in place of a single VertexBuffer.

Submeshes within a Mesh are rendered in the order of ascending Appearance layers, and such that opaque submeshes are rendered before transparent submeshes on the same layer. See the `setLayer` method in Appearance for more discussion on layered rendering.

Rendering and picking of a submesh is disabled if its Appearance is null.

## Deferred exceptions

An exception is thrown if the VertexBuffer or any of the IndexBuffers are in an invalid state when rendering or picking; see the respective class descriptions for more information. Note that it would be useless to check for these exception cases at construction time, because the application may freely change the contents of a VertexBuffer or an Appearance at any time. However, null IndexBuffers and VertexBuffers are blocked at the constructor, as usual, because the application is not able to change them afterwards.

**See Also:**
Binary format, `VertexBuffer`, `IndexBuffer`

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| `Mesh(VertexBuffer vertices, IndexBuffer[] submeshes, Appearance[] appearances)` Constructs a new Mesh with the given VertexBuffer and submeshes. |
| `Mesh(VertexBuffer vertices, IndexBuffer submesh, Appearance appearance)` Constructs a new Mesh consisting of only one submesh. |

## Method Summary

| | |
|---|---|
| `Appearance` | `getAppearance(int index)` Gets the current Appearance of the specified submesh. |
| `IndexBuffer` | `getIndexBuffer(int index)` Retrieves the submesh at the given index. |
| `int` | `getSubmeshCount()` Gets the number of submeshes in this Mesh. |
| `VertexBuffer` | `getVertexBuffer()` Gets the vertex buffer of this Mesh. |
| `void` | `setAppearance(int index, Appearance appearance)` Sets the Appearance for the specified submesh. |

| **Methods inherited from class javax.microedition.m3g.Node** |
|---|

| |
|---|
| `align`, `getAlignmentReference`, `getAlignmentTarget`, `getAlphaFactor`, `getParent`, `getScope`, `getTransformTo`, `isPickingEnabled`, `isRenderingEnabled`, `setAlignment`, `setAlphaFactor`, `setPickingEnable`, `setRenderingEnable`, `setScope` |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform`, `getOrientation`, `getScale`, `getTransform`, `getTranslation`, `postRotate`, `preRotate`, `scale`, `setOrientation`, `setScale`, `setTransform`, `setTranslation`, `translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack`, `animate`, `duplicate`, `find`, `getAnimationTrack`, `getAnimationTrackCount`, `getReferences`, `getUserID`, `getUserObject`, `removeAnimationTrack`, `setUserID`, `setUserObject` |

## Constructor Detail

### Mesh

```
public Mesh(VertexBuffer vertices,
            IndexBuffer submesh,
            Appearance appearance)
```

Constructs a new Mesh consisting of only one submesh. Rendering and picking of the Mesh is initially disabled if the Appearance is set to null.

**Parameters:**

`vertices` - a VertexBuffer to use for this mesh
`submesh` - an IndexBuffer defining the triangle strips to draw
`appearance` - an Appearance to use for this mesh, or null

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `submesh` is null

### Mesh

```
public Mesh(VertexBuffer vertices,
            IndexBuffer[] submeshes,
            Appearance[] appearances)
```

Constructs a new Mesh with the given VertexBuffer and submeshes. The number of submeshes is set equal to the length of the `submeshes` array. The `appearances` array is parallel to that, and must have at least as many elements. A null Appearance disables rendering and picking of the corresponding submesh. If the array itself is null, all appearances are initialized to null.

**Parameters:**

`vertices` - a VertexBuffer to use for all submeshes in this mesh
`submeshes` - an IndexBuffer array defining the submeshes to draw
`appearances` - an Appearance array parallel to `submeshes`, or null

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `submeshes` is null
`java.lang.NullPointerException` - if any element in `submeshes` is null
`java.lang.IllegalArgumentException` - if `submeshes` is empty
`java.lang.IllegalArgumentException` - if `(appearances != null) && (appearances.length < submeshes.length)`

## Method Detail

### setAppearance

```
public void setAppearance(int index,
                          Appearance appearance)
```

Sets the Appearance for the specified submesh.

**Parameters:**

`index` - index of the submesh to set the Appearance of
`appearance` - Appearance to set for the submesh at `index`, or null to disable the submesh

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getSubmeshCount)`

**See Also:**

`getAppearance`

### getAppearance

```
public Appearance getAppearance(int index)
```

Gets the current Appearance of the specified submesh.

**Parameters:**

`index` - index of the submesh to get the Appearance of

**Returns:**

current Appearance of the submesh at `index`

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getSubmeshCount)`

**See Also:**

`setAppearance`

### getIndexBuffer

```
public IndexBuffer getIndexBuffer(int index)
```

Retrieves the submesh at the given index.

**Parameters:**

`index` - index of the submesh to get

**Returns:**

current IndexBuffer at `index`

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getSubmeshCount)`

## getVertexBuffer

```
public VertexBuffer getVertexBuffer()
```

Gets the vertex buffer of this Mesh. This is always the original VertexBuffer that was supplied at construction. The VertexBuffer is never written to by the implementation. Specifically, the results of morphing (MorphingMesh) and skinning (SkinnedMesh) are *not* written to the VertexBuffer, nor are they exposed to the application by any other means.

In the case of a MorphingMesh, this VertexBuffer represents the base mesh. The morph target VertexBuffers can be retrieved with the `getMorphTarget` method in MorphingMesh.

**Returns:**

the (base) VertexBuffer of this Mesh

## getSubmeshCount

```
public int getSubmeshCount()
```

Gets the number of submeshes in this Mesh. This is always at least 1.

**Returns:**

the number of submeshes in this Mesh

**javax.microedition.m3g**

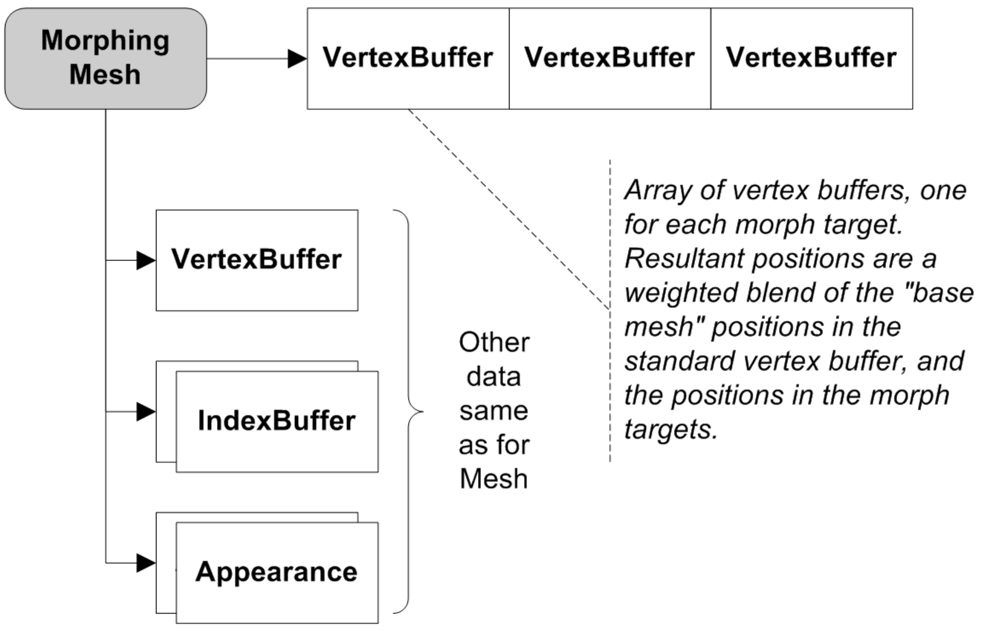
## Class MorphingMesh

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
          └ javax.microedition.m3g.Node
              └ javax.microedition.m3g.Mesh
                  └ javax.microedition.m3g.MorphingMesh
```

public class **MorphingMesh**
extends Mesh

A scene graph node that represents a vertex morphing polygon mesh.

MorphingMesh is equivalent to an ordinary Mesh, except that the vertices that are rendered are computed as a weighted linear combination of the base VertexBuffer and a number of *morph target* VertexBuffers. The resultant mesh is only used for rendering, and is not exposed to the application. The structure of a MorphingMesh object is shown in the figure below.


Morphing Mesh
VertexBuffer
VertexBuffer
VertexBuffer
Array of vertex buffers, one for each morph target. Resultant positions are a weighted blend of the "base mesh" positions in the standard vertex buffer, and the positions in the morph targets.
VertexBuffer
IndexBuffer
Appearance
Other data same as for Mesh

### Morph targets

All morph targets must have the same properties: The same types of arrays, the same number of vertices in each array, the same number of components per vertex, and the same component size. For example, it is prohibited for one morph target to contain vertex coordinates and texture coordinates, if some other target only contains vertex coordinates. Similarly, having 2D texture coordinates in one morph target and 3D texture coordinates in another is not allowed.

The base mesh must be a "superset" of the morph targets. If an array with certain type and dimensions exists in the morph targets, a similar array must also exist in the base mesh, but not vice versa. It is illegal, for example, for the morph targets to have per-vertex colors and 8-bit coordinates if the base mesh has 16-bit coordinates and/or no colors.

Only the VertexBuffer default color and the arrays that are present in the morph targets are actually morphed. The other arrays, as well as the scale and bias values, are copied from the base mesh. Scale and bias values of the morph targets are ignored.

## Morphing equation

Denoting the base mesh with **B**, the morph targets with $\mathbf{T_i}$, and the weights corresponding to the morph targets with $w_i$, the resultant mesh **R** is computed as follows:

$$\mathbf{R} = \mathbf{B} + \text{sum} [ w_i(\mathbf{T}_i - \mathbf{B}) ]$$

Any values for the weights $w_i$ are accepted, including negative values. The sum of the weights is similarly unconstrained. This allows having, for example, a model of a face with a neutral expression as the base mesh, and two morph targets where one is the base mesh but with a smiling mouth and the other with raised eyebrows. Now, setting the first weight to 1.0 makes the face smile, -0.5 could make it frown, and so on, while the eyebrow raising and lowering can be driven independent of the mouth movements.

Setting up the morph weights such that the individual weights as well as their sum are between [0, 1] ensures that the resultant mesh never grows beyond the convex hull of the base mesh and the targets. That, on the other hand, guarantees that no arithmetic overflows will occur and the results are as expected.

If the application chooses to set up the weights such that they or their sum is not in the [0, 1] interval, it should by some other means ensure that the morphed attributes of the resultant mesh will fit into the original numeric range, that is, in the same number of bits that are used in the base mesh and the morph targets. The available range for the results can be either [0, 255] or [0, 65535], for 8-bit and 16-bit components respectively. If the values do not, however, fit in that range, the results are undefined when rendering or picking.

Any intermediate values produced during morphing are subject to the dynamic range constraints that are specified in the package description. In other words, individual weights can be very large or small, as long as the resultant mesh fits in the 8/16-bit range.

The VertexBuffer scale and bias for the resultant mesh are taken from the base mesh as such, without interpolation, because correct interpolation between (integer) values that are in different (floating point) scales would require the interpolants to be first multiplied with the scale factor, and only then interpolated. This involves several floating point operations per vertex attribute, which would make morphing of anything but the most trivial meshes prohibitively expensive on current mobile hardware, for very little benefit. Note also that interpolating the scale terms separately from the values would not produce the correct results.

## Deferred exceptions

Any special cases and exceptions that are defined for Mesh also apply for MorphingMesh. An extra exception case is introduced due to the requirement that morph targets must be "subsets" of the base mesh, and that they must all have the same set of vertex attributes with the same dimensions, as specified above. This requirement cannot be enforced until when the morphing is actually done, that is, when rendering or picking.

**See Also:**

Binary format

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| `MorphingMesh(VertexBuffer base, VertexBuffer[] targets, IndexBuffer[] submeshes, Appearance[] appearances)` <br> Constructs a new MorphingMesh with the given base mesh and morph targets. |
| `MorphingMesh(VertexBuffer base, VertexBuffer[] targets, IndexBuffer submesh, Appearance appearance)` <br> Constructs a new MorphingMesh with the given base mesh and morph targets. |

## Method Summary

| | |
|---|---|
| `VertexBuffer` | `getMorphTarget(int index)` <br> Returns the morph target VertexBuffer at the given index. |
| `int` | `getMorphTargetCount()` <br> Returns the number of morph targets in this MorphingMesh. |
| `void` | `getWeights(float[] weights)` <br> Gets the current morph target weights for this mesh. |
| `void` | `setWeights(float[] weights)` <br> Sets the weights for all morph targets in this mesh. |

| **Methods inherited from class javax.microedition.m3g.Mesh** |
|---|
| `getAppearance, getIndexBuffer, getSubmeshCount, getVertexBuffer, setAppearance` |

| **Methods inherited from class javax.microedition.m3g.Node** |
|---|
| `align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope` |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|

`addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject`

## Constructor Detail

### MorphingMesh

```
public MorphingMesh(VertexBuffer base,
                    VertexBuffer[] targets,
                    IndexBuffer submesh,
                    Appearance appearance)
```

Constructs a new MorphingMesh with the given base mesh and morph targets. Except for the morph targets, the behavior of this constructor is identical to the corresponding constructor in Mesh; refer to that for more information.

The morph target weights are initially set to zero, meaning that the resultant mesh is equal to the base mesh. The behavior of a newly constructed MorphingMesh is therefore equivalent to an ordinary Mesh.

**Parameters:**

`base` - a VertexBuffer representing the base mesh
`targets` - a VertexBuffer array representing the morph targets
`submesh` - an IndexBuffer defining the triangle strips to draw
`appearance` - an Appearance to use for this mesh, or null

**Throws:**

`java.lang.NullPointerException` - if `base` is null
`java.lang.NullPointerException` - if `targets` is null
`java.lang.NullPointerException` - if `submesh` is null
`java.lang.NullPointerException` - if any element in `targets` is null
`java.lang.IllegalArgumentException` - if `targets` is empty

### MorphingMesh

```
public MorphingMesh(VertexBuffer base,
                    VertexBuffer[] targets,
                    IndexBuffer[] submeshes,
                    Appearance[] appearances)
```

Constructs a new MorphingMesh with the given base mesh and morph targets. Except for the morph targets, the behavior of this constructor is identical to the corresponding constructor in Mesh; refer to that for more information.

The morph target weights are initially set to zero, meaning that the resultant mesh is equal to the base mesh. The behavior of a newly constructed MorphingMesh is therefore equivalent to an ordinary Mesh.

**Parameters:**

`base` - a VertexBuffer representing the base mesh
`targets` - a VertexBuffer array representing the morph targets
`submeshes` - an IndexBuffer array defining the submeshes to draw
`appearances` - an Appearance array parallel to `submeshes`, or null

**Throws:**

`java.lang.NullPointerException` - if `base` is null
`java.lang.NullPointerException` - if `targets` is null
`java.lang.NullPointerException` - if `submeshes` is null
`java.lang.NullPointerException` - if any element in `targets` is null
`java.lang.NullPointerException` - if any element in `submeshes` is null
`java.lang.IllegalArgumentException` - if `targets` is empty
`java.lang.IllegalArgumentException` - if `submeshes` is empty
`java.lang.IllegalArgumentException` - if `(appearances != null) && (appearances.length < submeshes.length)`

## Method Detail

### getMorphTarget

`public VertexBuffer getMorphTarget(int index)`

Returns the morph target VertexBuffer at the given index.

**Parameters:**
`index` - the index of the morph target to get

**Returns:**
the VertexBuffer object at `index`

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `(index < 0) || (index >= getMorphTargetCount)`

**See Also:**
`Mesh.getVertexBuffer`

### getMorphTargetCount

`public int getMorphTargetCount()`

Returns the number of morph targets in this MorphingMesh.

**Returns:**
the number of morph targets

### setWeights

`public void setWeights(float[] weights)`

Sets the weights for all morph targets in this mesh. The number of weights copied in is the number of target vertex buffers, as specified at construction time. The source array must have at least that many elements. See the class description for more information.

**Parameters:**

`weights` - weight factors for all morph targets

**Throws:**

`java.lang.NullPointerException` - if `weights` is null

`java.lang.IllegalArgumentException` - if `weights.length < getMorphTargetCount`

**See Also:**

`getWeights`

## getWeights

```
public void getWeights(float[] weights)
```

Gets the current morph target weights for this mesh.

**Parameters:**

`weights` - array to be populated with the morph target weights

**Throws:**

`java.lang.NullPointerException` - if `weights` is null

`java.lang.IllegalArgumentException` - if `weights.length < getMorphTargetCount`

**See Also:**

`setWeights`

**javax.microedition.m3g**

## Class Node

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
          └ javax.microedition.m3g.Node
```

**Direct Known Subclasses:**
Camera, Group, Light, Mesh, Sprite3D

public abstract class **Node**
extends Transformable

An abstract base class for all scene graph nodes.

There are five different kinds of nodes:

- `Camera` defines the projection from 3D to 2D, as well as the position of the viewer in the scene.
- `Mesh` defines a 3D object, consisting of triangles with associated material properties.
- `Sprite3D` defines a screen-aligned 2D image with a position in 3D space.
- `Light` defines the position, direction, color and other attributes of a light source.
- `Group` serves as a root for scene graph branches.

### Node transformation

Each node defines a local coordinate system that can be transformed relative to the coordinate system of the parent node. The transformation *from* the local coordinate system of a node *to* the coordinate system of its parent is called the *node transformation*.

The node transformation consists of four parts: a generic matrix **M**, a non-uniform scale **S**, an orientation **R** and a translation **T**. The bottom row of **M** must be equal to (0 0 0 1). The methods to manipulate the individual components are defined in the base class, Transformable.

To transform a point from a node's local coordinates to its parent's coordinates, the point is multiplied by the transformation components in the order that they are listed above. Formally, a homogeneous vector **p** = (x, y, z, 1), representing a 3D point in the local coordinate system, is transformed into **p**' = (x', y', z', 1) in the parent coordinate system as follows:

$$\mathbf{p}' = \mathbf{T}\ \mathbf{R}\ \mathbf{S}\ \mathbf{M}\ \mathbf{p}$$

The translation, orientation and scale components of the node transformation can be animated independently from each other. The matrix component is not animatable at all; it can only be changed using the `setTransform` method.

## Node alignment

A node may be *aligned* with respect to a selected *reference node* (or nodes). This means that the aligned node is, upon request, automatically oriented so that its coordinate system matches the reference node's coordinate system in the specified way. A common use case for node alignment is to create "billboards" that are always facing the camera; another is to make the camera always point at a certain object.

When a node is aligned, its original orientation component **R** is overwritten with an aligned orientation **A**. (The aligned orientation is computed as specified below, in the Implementation Guidelines section.) The other components of the node transformation are not affected by alignment. The transformation from the local coordinate system of an aligned node to its parent node's coordinate system is, therefore,

$$\mathbf{p}' = \mathbf{T\ A\ S\ M\ p}$$

The application must explicitly call the `align` method on a node (or any of its ancestors) when it requires the alignments of that node and its descendants to be computed. This is typically done once per frame, before rendering. Rendering operations do not resolve any alignments; they simply use whatever orientation each node has at that time. The same holds true for `getTransformTo` and any other methods whose results depend on the orientation.

The alignment reference node(s) and the method of alignment are selected with `setAlignment`. This does not yet compute the new aligned orientation, but merely specifies how that is to be done. Optionally, the reference node may be left unspecified (null) until when `align` is called; the reference node is then supplied as a parameter to `align`. This is very useful for billboards, because otherwise the application would have to call `setAlignment` separately for every billboard in the scene whenever the camera is changed.

## Inherited node properties

Besides the node transformation, there are three node properties whose effective values are in some manner influenced by the ancestors of each node. These properties are the alpha factor, the rendering enable flag, and the picking enable flag.

The *alpha factor* allows (groups of) Mesh and Sprite3D objects to be faded in and out in a convenient way, provided that certain preconditions related to their Appearance are met. The alpha factor is defined for each Node, and its value is between [0, 1]. The effective alpha factor for an object is obtained by multiplying its local alpha factor with the alpha factors of its ancestors. The alpha factor is ignored for Light and Camera nodes.

When rendering a Mesh, its effective alpha factor is multiplied with the alpha component of the diffuse color in each of the Material objects associated with that Mesh. In absence of a Material object, the alpha factor is applied to the alpha channel of the VertexBuffer color array, or if the color array is null, the default color alpha component. When rendering a Sprite3D, its effective alpha factor is multiplied with the alpha channel of the sprite image. Note that for both meshes and sprites, only the alpha values are ever modified. The alpha factor alone is therefore not sufficient for a fade-in/fade-out effect. Instead, the texture blending mode, the framebuffer blending mode, and the alpha threshold must all be set appropriately. For meshes, setting texture blending to `MODULATE`, framebuffer blending to `ALPHA`, and alpha threshold to zero will often produce the desired result. Sprites should use a non-zero alpha threshold and `ALPHA` blending in CompositingMode.

The *enable flags* for rendering and picking allow (groups of) mesh and sprite objects to be made "invisible" from the point of view of rendering and picking, respectively. The effective enable status of a node is the logical AND of the enable flags on that node and all its ancestors. Therefore, setting the enable flag of a node to *true* does not guarantee that the node will be rendered or picked. Rather, if any of its ancestors are disabled, the node will be ignored regardless of its own enable flag.

Note that the scope of a Node is *not* an inherited property; see below for more information.

## Scoping

The *scope* of a Node is an integer bitmask that allows scene graph nodes to form conceptual groups independent of the scene graph hierarchy. In other words, nodes that are in a particular Group are not necessarily in the same scope. Formally, two nodes A and B are defined to be in the same scope if the bitwise AND of their scopes is non-zero:

$scope_A$ & $scope_B$ != 0

Scopes are not hierarchic in any way. In particular, the scope of a Group or SkinnedMesh node is not propagated to or inherited by its children. After all, scopes are intended to be separate from the scene hierarchy.

Scoping serves three purposes:

- **Visibility culling**. Only those objects are rendered that are in the same scope as the Camera. This gives an additional means to control the set of visible objects, complementary to the rendering enable flag.

- **Lighting**. A light source only has an effect on Meshes that are in the same scope with it. This makes it possible to have a very large number of light sources in a scene graph without having all the lights illuminate all meshes. Besides being impractical, that would also be prohibitively expensive in terms of processing power.

- **Picking.** The scope of the pick ray is given as a parameter to the `pick` methods in Group. Again, only those objects can be picked that are in the same scope as the pick ray; the others are ignored.

The default scope is `-1`, implying that all nodes are in the same scope. By default, all objects are therefore visible to all cameras, and are lit by all light sources.

## Instantiation

Node is an abstract class, and therefore has no public constructor. When a class derived from Node is instantiated, the attributes defined in Node will have the following default values:

- parent node : null
- rendering enable : *true*
- picking enable : *true*
- alpha factor : 1.0
- scope : `-1`
- alignment : ( `NONE`, null ) for all axes

## Implementation guidelines

The alignment rotation **A** is computed relative to the initial coordinate system A defined by the **T** component of the node transformation alone. All other transformation components of the node being aligned are ignored.

Conceptually, alignment is composed of two cumulative rotations: the shortest rotation $\mathbf{R}_z$ that takes the initial Z axis to the Z alignment target vector, followed by the rotation $\mathbf{R}_y$ about the resulting Z vector that minimizes the angle between the resulting Y axis and the Y alignment target vector. If alignment is set for one axis only, that rotation is performed like the initial Z rotation.

Formally, let us denote by $\mathbf{t}_Z$ and $\mathbf{t}_Y$ the Z and Y alignment target vectors, transformed from their respective reference nodes to A; note that axis targets transform as vectors, and origin targets as points. The axis for the first rotation $\mathbf{R}_z$ is then the cross product of the local Z axis of A and the target vector:

$$\mathbf{a}_Z = (0\ 0\ 1)^T \times \mathbf{t}_Z$$

and the rotation angle can be computed via the dot product of the two. Rotating by $\mathbf{R}_z$ takes us to a new coordinate frame B where $\mathbf{t}_Y$ is expressed as:

$$\mathbf{t}_Y' = \mathbf{R}_Z^{-1} \times \mathbf{t}_Y$$

The axis for the second rotation $\mathbf{R}_Y$ is the local Z axis of B, and the angle is the angle between the local Y axis and the projection of $\mathbf{t}_Y'$ on the XY plane. The final alignment rotation **A** is then:

$$\mathbf{A} = \mathbf{R}_Z\ \mathbf{R}_Y$$

There are two cases where a rotation axis is undefined. Firstly, if either target vector coincides with the axis that it is a target for, the respective rotation must be substituted with an identity rotation. Secondly, if the target vector and the axis are opposite, the exact rotation path (that is, the resultant direction of the other two axes) is implementation dependent, but must be deterministic. Note that the latter only matters for unconstrained (single-axis) alignment.

**See Also:**
Binary format

## Field Summary

| | |
|---|---|
| `static int` | **`NONE`**<br>Specifies for the `setAlignment` method that no alignment should be done for the specified axis. |
| `static int` | **`ORIGIN`**<br>Specifies the origin of the reference node as an orientation reference for the `setAlignment` method. |
| `static int` | **`X_AXIS`**<br>Specifies the X axis of the reference node as an orientation reference for the `setAlignment` method. |
| `static int` | **`Y_AXIS`**<br>Specifies the Y axis of the reference node as an orientation reference for the `setAlignment` method. |
| `static int` | **`Z_AXIS`**<br>Specifies the Z axis of the reference node as an orientation reference for the `setAlignment` method. |

## Method Summary

| | |
|---|---|
| `void` | **`align`**`(Node reference)`<br>Applies alignments to this Node and its descendants. |

| | |
|---|---|
| `Node` | `getAlignmentReference(int axis)` Returns the alignment reference node for the given axis. |
| `int` | `getAlignmentTarget(int axis)` Returns the alignment target for the given axis. |
| `float` | `getAlphaFactor()` Retrieves the alpha factor of this Node. |
| `Node` | `getParent()` Returns the scene graph parent of this node. |
| `int` | `getScope()` Retrieves the scope of this Node. |
| `boolean` | `getTransformTo(Node target, Transform transform)` Gets the composite transformation from this node to the given node. |
| `boolean` | `isPickingEnabled()` Retrieves the picking enable flag of this Node. |
| `boolean` | `isRenderingEnabled()` Retrieves the rendering enable flag of this Node. |
| `void` | `setAlignment(Node zRef, int zTarget, Node yRef, int yTarget)` Sets this node to align with the given other node(s), or disables alignment. |
| `void` | `setAlphaFactor(float alphaFactor)` Sets the alpha factor for this Node. |
| `void` | `setPickingEnable(boolean enable)` Sets the picking enable flag of this Node. |
| `void` | `setRenderingEnable(boolean enable)` Sets the rendering enable flag of this Node. |
| `void` | `setScope(int scope)` Sets the scope of this node. |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Field Detail

**NONE**

```
public static final int NONE
```

Specifies for the `setAlignment` method that no alignment should be done for the specified axis.

**See Also:**
Constant Field Values

### ORIGIN

```
public static final int ORIGIN
```

Specifies the origin of the reference node as an orientation reference for the `setAlignment` method.

**See Also:**
Constant Field Values

### X_AXIS

```
public static final int X_AXIS
```

Specifies the X axis of the reference node as an orientation reference for the `setAlignment` method.

**See Also:**
Constant Field Values

### Y_AXIS

```
public static final int Y_AXIS
```

Specifies the Y axis of the reference node as an orientation reference for the `setAlignment` method.

**See Also:**
Constant Field Values

### Z_AXIS

```
public static final int Z_AXIS
```

Specifies the Z axis of the reference node as an orientation reference for the `setAlignment` method.

**See Also:**
Constant Field Values

## Method Detail

### setRenderingEnable

`public void **setRenderingEnable**(boolean enable)`

Sets the rendering enable flag of this Node. The effective rendering enable status for this node is the logical AND of the enable flags on this node and all its ancestors. Therefore, the node is disabled if any of its ancestors are. The node's own status has an effect only if all the ancestors are enabled.

If the effective status is *true*, this node is enabled for rendering; otherwise, it is disabled. Sprite3D, Mesh and Light nodes are turned on and off with this setting, but on Camera nodes it is ignored.

**Parameters:**
`enable` - *true* to enable rendering; *false* to disable

### setPickingEnable

`public void **setPickingEnable**(boolean enable)`

Sets the picking enable flag of this Node. The effective picking enable status for this node is the logical AND of the enable flags on this node and all its ancestors. Therefore, the node is disabled if any of its ancestors are. The node's own status has an effect only if all the ancestors are enabled.

If the effective status is *true*, this node is enabled for picking; otherwise, it is disabled. This setting is ignored for Lights and Cameras, because they are unpickable in any case.

**Parameters:**
`enable` - *true* to enable picking; *false* to disable

### setScope

`public void **setScope**(int scope)`

Sets the scope of this node. The scope is used to limit the set of nodes that are taken into account in rendering, lighting and picking. See the class description for more information.

**Parameters:**
`scope` - the new scope for this node
**See Also:**
`getScope`

### setAlphaFactor

`public void **setAlphaFactor**(float alphaFactor)`

Sets the alpha factor for this Node. This can be used to fade groups of meshes and sprites in and out. The alpha factor has no effect on Light and Camera nodes. See the class description for more information.

**Parameters:**

`alphaFactor` - the new alpha factor for this node; must be [0, 1]

**Throws:**

`java.lang.IllegalArgumentException` - if `alphaFactor` is negative or greater than 1.0

**See Also:**

`getAlphaFactor`

## isRenderingEnabled

```
public boolean isRenderingEnabled()
```

Retrieves the rendering enable flag of this Node. Note that this is not the effective rendering enable status, but only the local status of this Node.

**Returns:**

the rendering enable flag of this Node

**See Also:**

`setRenderingEnable`

## isPickingEnabled

```
public boolean isPickingEnabled()
```

Retrieves the picking enable flag of this Node. Note that this is not the effective picking enable status, but only the local status of this Node.

**Returns:**

the picking enable flag of this Node

**See Also:**

`setPickingEnable`

## getScope

```
public int getScope()
```

Retrieves the scope of this Node.

**Returns:**

the current scope of this Node

**See Also:**

`setScope`

## getAlphaFactor

```
public float getAlphaFactor()
```

Retrieves the alpha factor of this Node. Note that this is not the effective alpha factor, but only the local alpha factor of this Node. To put it another way, the alpha factors of any ancestors to this Node are not multiplied in.

**Returns:**

the alpha factor of this node; [0, 1]

**See Also:**

`setAlphaFactor`

## getParent

```
public Node getParent()
```

Returns the scene graph parent of this node.

**Returns:**

reference to the parent node, or null if there is no parent

## getTransformTo

```
public boolean getTransformTo(Node target,
                              Transform transform)
```

Gets the composite transformation from this node to the given node. The composite transformation is defined to be such that it transforms a point in the local coordinate system of this node to the coordinate system of the given node. For example, the composite transformation from this node to its parent is equal to the node transformation of this node. Similarly, the composite transformation from this node to its child is equal to the inverse of the node transformation of the child.

If there is no path from this node to the given node, this method returns *false*. On the other hand, if there is a path but the transformation cannot be computed due to a singular transformation, an ArithmeticException is thrown. Beware that a transformation that is invertible in one implementation may not be invertible in another, because of different arithmetic accuracy. To be safe, avoid matrix elements with very small or very large absolute values. See also the package description.

**Parameters:**

`target` - transformation target node

`transform` - transform object to receive the transformation; if there is no path to the target node, the contents of the object are left undefined

**Returns:**

*true* if the returned transformation is valid; *false* if there is no path from this node to the target node

**Throws:**

`java.lang.NullPointerException` - if `target` is null

`java.lang.NullPointerException` - if `transform` is null

`java.lang.ArithmeticException` - if the inverse of a transformation along the path is required, but can not be computed

## align

```
public final void align(Node reference)
```

Applies alignments to this Node and its descendants.

The aligned orientation for this node and all its descendants are calculated in an undefined order. The rare case where there are chains of dependencies between aligned objects is therefore not necessarily taken into account.

The orientation component of the node transformation of each aligned node is overwritten with the aligned orientation. The pre-existing orientation is not preserved.

A reference node can be passed in to this method, in order to allow alignment of objects to a common reference that is determined at run time. This is usually used to align items to the active camera, for use as billboards or impostors. Since the active camera can change, a reference to it cannot be directly encoded in the scene graph. Instead, it is passed in as an argument to this method.

See the class description and `setAlignment` for more information on how to set up and apply alignments.

**Parameters:**

`reference` - a node to serve as a common alignment reference for nodes that have no fixed reference in either or both axes, or null to use this node as the common reference

**Throws:**

`java.lang.IllegalArgumentException` - if `reference` is not in the same scene graph as this node

`java.lang.IllegalStateException` - if the `zRef` or `yRef` node of any aligned node is not in the same scene graph as the aligned node

`java.lang.IllegalStateException` - if any node is aligned to itself or its descendant (note: this applies to null alignment references, as well)

`java.lang.ArithmeticException` - if a transformation required in the alignment computations cannot be computed

## setAlignment

```
public void setAlignment(Node zRef,
                         int zTarget,
                         Node yRef,
                         int yTarget)
```

Sets this node to align with the given other node(s), or disables alignment. Alignment can be used, for example, for automatic "look at" behavior for the camera or a spot light, and to create "billboards" that are always facing the active camera directly.

Alignment can be set or disabled for one or both of the Y and Z axes. If it is set for both, the Z alignment is applied first, followed by the Y alignment. The Y alignment is constrained by the Z alignment. If alignment is set for one axis only, it is unconstrained.

Alignment can be disabled for either or both axes by setting the respective alignment targets to `NONE`. If both alignments are disabled, the orientation is left at its present state. The original unaligned orientation is *not* restored.

**Parameters:**

`zRef` - the node to use as reference for aligning the Z axis of this node, or null to use instead the reference node passed as an argument to the `align` method

`zTarget` - the axis of `zRef` to align the Z axis of this node with, or `ORIGIN` to have the Z axis point at the origin of `zRef`, or `NONE` to not align the Z axis at all

`yRef` - the Y axis equivalent of `zRef`

`yTarget` - the Y axis equivalent of `zTarget`

**Throws:**

`java.lang.IllegalArgumentException` - if `yTarget` or `zTarget` is not one of the

symbolic constants listed above
`java.lang.IllegalArgumentException` - if `(zRef == yRef) && (zTarget == yTarget != NONE)`
`java.lang.IllegalArgumentException` - if `zRef` or `yRef` is this Node

**See Also:**
`align`, `getAlignmentTarget`, `getAlignmentReference`

**Example:**
**Common use cases for node alignment.**

```
setAlignment(null, Node.NONE, null, Node.NONE);         // Disabled
 setAlignment(null, Node.Z_AXIS, null, Node.Y_AXIS);     // "Sprite"
 setAlignment(null, Node.ORIGIN, world, Node.Y_AXIS);    // Billboard
 setAlignment(target, Node.ORIGIN, target, Node.NONE);   // Target light
 setAlignment(target, Node.ORIGIN, world, Node.Y_AXIS);  // Target camera

 // NOTE 1:
 // The billboard alignment example requires that world space "up"
 // is Y and billboard space "up" is Z, so that the Z alignment is
 // constrained by the Y alignment and not vice versa. Otherwise,
 // the billboard will not stand upright as the camera passes from
 // above or below; instead, it will lean over and eventually lie
 // flat on the ground. The M component of the billboard's node
 // transformation can be used to rotate the billboard into the
 // right orientation; the R component can not, because it gets
 // overwritten by the aligned orientation. Another option is to
 // use an extra Group node.

 // NOTE 2:
 // A camera or light is always facing towards its negative Z axis
 // in its local coordinate system. To make the target camera and
 // light alignments work as expected, the Z axis must be made to
 // point in the opposite direction. This can be done by rotating
 // the node 180 degrees about its local Y axis. This, in turn,
 // can be done as described in Note 1 (above), or somewhat more
 // conveniently, using the scale (S) component:

 camera.scale(-1, 1, -1);    // rotate 180 degrees about the Y axis
```

## getAlignmentTarget

`public int getAlignmentTarget(int axis)`

Returns the alignment target for the given axis.

**Parameters:**
`axis` - the node axis to query the target for; one of `Y_AXIS` and `Z_AXIS`

**Returns:**
the alignment target; one of the symbolic constants allowed for the `zTarget` and `yTarget` parameters of `setAlignment`

**Throws:**
`java.lang.IllegalArgumentException` - if `axis` is not one of the symbolic constants listed for `axis` above

**Since:**
M3G 1.1

**See Also:**
`setAlignment`, `align`

## getAlignmentReference

`public Node **getAlignmentReference**(int axis)`

Returns the alignment reference node for the given axis.

Note that alignment reference nodes are *not* returned in a call to `getReferences`.

**Parameters:**
`axis` - the node axis to query the reference node for; one of `Y_AXIS` and `Z_AXIS`

**Returns:**
the alignment reference node

**Throws:**
`java.lang.IllegalArgumentException` - if `axis` is not one of the symbolic constants listed for `axis` above

**Since:**
M3G 1.1

**See Also:**
`setAlignment`, `align`

**javax.microedition.m3g**

# Class Object3D

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
```

**Direct Known Subclasses:**

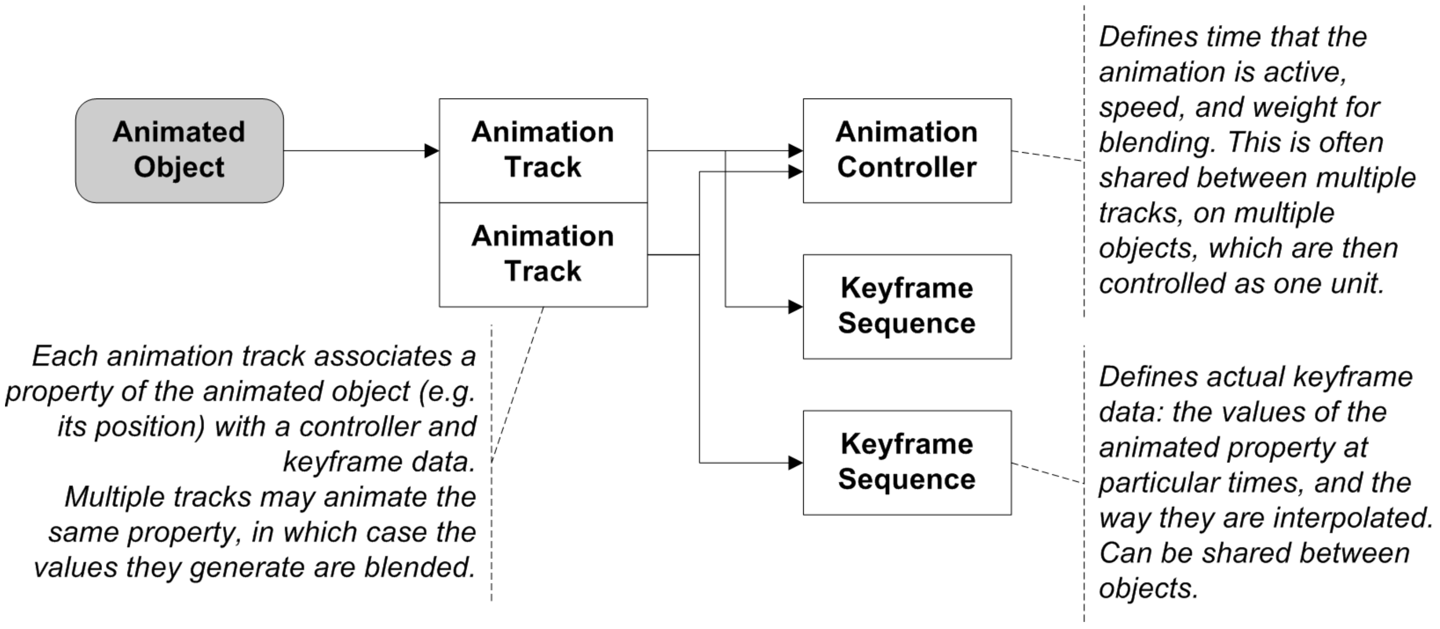
AnimationController, AnimationTrack, Appearance, Background, CompositingMode, Fog, Image2D, IndexBuffer, KeyframeSequence, Material, PolygonMode, Transformable, VertexArray, VertexBuffer

public abstract class **Object3D**
extends java.lang.Object

An abstract base class for all objects that can be part of a 3D world. This includes the world itself, other scene graph nodes, animations, textures, and so on. In fact, everything in this API is an Object3D, except for Loader, Transform, RayIntersection, and Graphics3D.

## Animation

Animations are applied to an object and its descendants with the `animate` method in this class. The objects needed for animation and their relationships are shown in the figure below.


Animated Object
Animation Track
Animation Track
Animation Controller
Keyframe Sequence
Keyframe Sequence
Defines time that the animation is active, speed, and weight for blending. This is often shared between multiple tracks, on multiple objects, which are then controlled as one unit.
Each animation track associates a property of the animated object (e.g. its position) with a controller and keyframe data. Multiple tracks may animate the same property, in which case the values they generate are blended.
Defines actual keyframe data: the values of the animated property at particular times, and the way they are interpolated. Can be shared between objects.

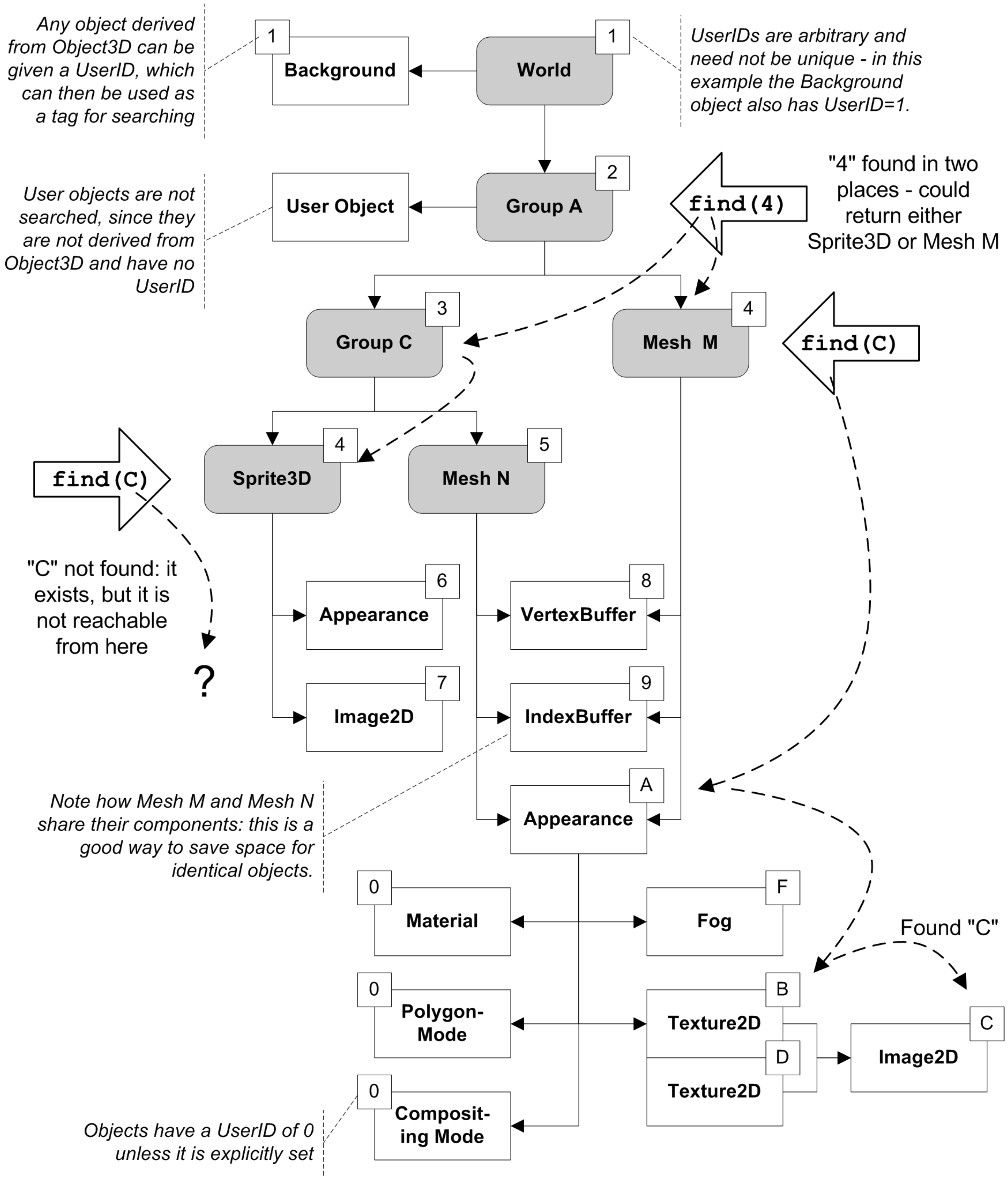
## Finding objects

Every Object3D can be assigned a *user ID*, either at authoring stage or at run time with the `setUserID` method. User IDs are typically used to find a known object in a scene loaded from a data stream.

The `find` method searches through all objects that are *reachable* from this object through a chain of references, and returns the one with the given user ID. If there are multiple objects with the same ID, the implementation may return any one of them.

An object **O** is defined to be reachable from itself and from all objects that have a chain of direct references to it. The parent reference and the alignment references in a Node do not count as direct references.

The operation of `find` is illustrated in the figure below.


Any object derived from Object3D can be given a UserID, which can then be used as a tag for searching
1
Background
1
World
UserIDs are arbitrary and need not be unique - in this example the Background object also has UserID=1.
User objects are not searched, since they are not derived from Object3D and have no UserID
User Object
2
Group A
find(4)
"4" found in two places - could return either Sprite3D or Mesh M
3
Group C
4
Mesh M
find(C)
find(C)
4
Sprite3D
5
Mesh N
"C" not found: it exists, but it is not reachable from here
?
6
Appearance
7
Image2D
8
VertexBuffer
9
IndexBuffer
A
Appearance
Note how Mesh M and Mesh N share their components: this is a good way to save space for identical objects.
0
Material
F
Fog
Found "C"
0
Polygon-Mode
B
Texture2D
D
Texture2D
C
Image2D
0
Composit-ing Mode
Objects have a UserID of 0 unless it is explicitly set

### Associated user data

Object3D has an attribute called the *user object*. The user object may contain any arbitrary Object, whose interpretation and usage are entirely up to each application. The user object is held by reference, and its contents are never accessed by the implementation.

If an Object3D is loaded from a file by the Loader, the user object may contain a Hashtable that stores byte array values keyed by Integers. This is the case when one or more *user parameters* are associated with the serialized Object3D; see also the file format specification. If there are no user parameters, the user object is initially set to null.

A typical example of using this type of persistent user data is to include application parameters inside a scene graph, such as in multi-level games, where a non-player character may have a series of attributes such as hit strength, armor, initial health, and so on. Although it is possible to have this information in a separate file, it is neater, easier and less error prone to associate it directly with the Object3D that represents the character.

### Instantiation

Object3D is an abstract class, and therefore has no public constructors. When a class derived from Object3D is instantiated, the attributes inherited from Object3D will have the following default values:

- user ID : 0
- user object : null
- animation tracks : none

**See Also:**
Binary format

| **Method Summary** | |
|---|---|
| void | **addAnimationTrack**(AnimationTrack animationTrack)<br>Adds the given AnimationTrack to this Object3D, potentially changing the order and indices of the previously added tracks. |
| int | **animate**(int time)<br>Updates all animated properties in this Object3D and all Object3Ds that are reachable from this Object3D. |
| Object3D | **duplicate**()<br>Creates a duplicate of this Object3D. |
| Object3D | **find**(int userID)<br>Retrieves an object that has the given user ID and is reachable from this object. |
| AnimationTrack | **getAnimationTrack**(int index)<br>Gets an AnimationTrack by index. |
| int | **getAnimationTrackCount**()<br>Gets the number of AnimationTracks currently associated with this Object3D. |
| int | **getReferences**(Object3D[] references)<br>Returns the number of direct Object3D references in this object, and fills in the objects to the given array. |

| | |
|---|---|
| `int` | `getUserID()`<br>Gets the user ID of this object. |
| `java.lang.Object` | `getUserObject()`<br>Retrieves the user object that is currently associated with this Object3D. |
| `void` | `removeAnimationTrack(AnimationTrack animationTrack)`<br>Removes the given AnimationTrack from this Object3D, potentially changing the order and indices of the remaining tracks. |
| `void` | `setUserID(int userID)`<br>Sets the user ID for this object. |
| `void` | `setUserObject(java.lang.Object userObject)`<br>Associates an arbitrary, application specific Object with this Object3D. |

## Method Detail

### animate

`public final int` **`animate`**`(int time)`

Updates all animated properties in this Object3D and all Object3Ds that are reachable from this Object3D. Objects that are not reachable are not affected. See the class description for the definition of reachability.

Animated properties are set to their interpolated values pertaining to the time given as a parameter. The pre-existing values of the target properties are *overwritten* with the animated values, discarding the pre-existing values. The original values are *not* restored even if the animation is terminated.

The unit of time used in animation is defined by the application and does not have to correspond to real time in any way. Importantly, the animation system does not need to know what the time unit is. Milliseconds are often used by convention, but any other unit is equally valid.

If a property is targeted by an AnimationTrack, but the AnimationTrack is not associated with an active AnimationController, the property is not updated.

Typically, the application would call this method once per frame, with strictly increasing values of `time`. For example, if the application wishes to draw 20 frames per second, the value of `time` should be increased by 50 between successive calls, assuming a time unit of one millisecond.

Even though strictly increasing values of `time` are often used, this is not a requirement. The application can pass in any time value. To put it another way, the animation system supports random access. This allows the application to, for example, rewind or restart the animations easily.

In order to allow the application to throttle the frame rate depending on the characteristics of the animation, this method returns a *validity interval*. This is the amount of time for which the active animations on this object are guaranteed to make no changes to the reachable animated objects. For an object with no references to other animatable objects, this is determined solely from its own animation information. Otherwise, it is the minimum of this, and the returned validity interval of all reachable animated objects.

For example, consider a single object with an active animation that starts to change an object's properties only at t=1000. If we call animate() at t=500, the validity interval returned should ideally be 500. The application can, in the absence of external events, then choose not to render another frame for 500 time units. This estimate of validity must be conservative, so it is acceptable (but not friendly) for an implementation to always return 0 from this method.

If no animations are active on this object, the fact that a conservative estimate is required permits any interval to be returned, but it is strongly recommended that the value in this case should correspond to the maximum positive integer.

**Parameters:**
`time` - world time to update the animations to
**Returns:**
validity interval; the number of time units until this method needs to be called again for this or any reachable Object3D
**Throws:**
`java.lang.IllegalStateException` - if any active animation violates the constraints defined in KeyframeSequence
**See Also:**
`KeyframeSequence`, `AnimationController`, `AnimationTrack`

## duplicate

`public final Object3D duplicate()`

Creates a duplicate of this Object3D.

Duplication has no effect on this object or any other existing object; it merely creates one or more new objects.

As a general rule, a duplicate object will have exactly the same properties as the original object, including attribute values, references to other objects, and any other contained data. However, if this object is a Node, duplication is done as follows:

1. This Node is always copied. The parent of the duplicate Node is set to null.

2. Any descendants of this Node are themselves duplicated; this includes the skeleton group of a SkinnedMesh. Any other referenced objects are *not* duplicated.

3. If any Node in the duplicate set of Nodes refers to any Node in the original set, then that reference is updated to the corresponding Node in the duplicate set. All other references are left as they are, even if this results in a scene graph branch that is in an illegal state.

Note that the duplicate object will also have the same user ID as the original. The application is responsible for assigning the IDs in the first place, so setting the ID of the duplicate object to some unique value, if so desired, is also the application's responsibility.

Duplication is not supported for user defined classes. That is, if the application extends any class defined in this API, any instances of that class will be treated by this method as instances of the base class. For example, duplicating an instance of MonsterMesh (derived from Mesh) will produce just a Mesh instance, not a MonsterMesh instance.

This method is similar to the `clone` method that is available on the higher end Java platforms, such as J2SE and J2ME/CDC. This method is likely to be deprecated once the proper `java.lang.Object.clone` method becomes available also on CLDC.

**Returns:**
a new Object3D that is a duplicate of this object

## find

```
public Object3D find(int userID)
```

Retrieves an object that has the given user ID and is reachable from this object. If there are multiple objects with the same ID, the implementation may return any one of them. See the class description for the definition of reachability.

**Parameters:**
`userID` - the user ID to search for
**Returns:**
the first object encountered that has the given user ID, or null if no matching objects were found

## getReferences

```
public int getReferences(Object3D[] references)
```

Returns the number of direct Object3D references in this object, and fills in the objects to the given array. If the array is null, only the number of references is returned. Duplicate references are explicitly not eliminated, that is, the same object may appear multiple times in the array.

The parent reference and the alignment references in a Node do not count as direct references, and are hence not returned. Also, null references are never returned.

This method is provided to facilitate scene graph traversal, which is otherwise a non-trivial operation; tracking the links from one object to another requires different code for each type of object. Typical usage of this method is to first call it with a null array, then use the number of references returned to allocate or resize the target array, and then call again with the actual target array. This is illustrated in the example below.

**Parameters:**
`references` - an array of Object3D references to be filled in, or null to only return the number of references
**Returns:**
the number of direct Object3D references in this object (note: the number of *unique* references may be smaller)
**Throws:**
`java.lang.IllegalArgumentException` - if `(references != null) && (references.length < getReferences(null))`
**Example:**
**A recursive method to traverse all descendants of an object.**

```
void traverseDescendants(Object3D obj)
  {
     int numReferences = obj.getReferences(null);
```

```
    if (numReferences > 0)
    {
       Object3D[] objArray = new Object3D[numReferences];
       obj.getReferences(objArray);
       for (int i = 0; i < numReferences; i++)
       {
          processObject(objArray[i]);       // process object i...
          traverseDescendants(objArray[i]); // ...and its descendants
       }
    }
 }
```

## setUserID

`public void **setUserID**(int userID)`

Sets the user ID for this object.

**Parameters:**
`userID` - the ID to set

**See Also:**
`getUserID()`

## getUserID

`public int **getUserID**()`

Gets the user ID of this object.

**Returns:**
the current user ID

**See Also:**
`setUserID(int)`

## setUserObject

`public void **setUserObject**(java.lang.Object userObject)`

Associates an arbitrary, application specific Object with this Object3D. The given user object replaces any previously set object. See the class description for more information.

The user object is stored by reference. Its contents are never accessed by the implementation, but the reference is copied in the `duplicate` operation.

**Parameters:**
`userObject` - the Object to associate with this Object3D, or null to remove any existing association

**See Also:**
`getUserObject`

## getUserObject

`public java.lang.Object` **`getUserObject`**`()`

Retrieves the user object that is currently associated with this Object3D. If an Object3D is constructed by the Loader, the user object may initially be a Hashtable containing persistent user data in the form of byte arrays keyed by Integers.

**Returns:**
the current user object associated with this Object3D

**See Also:**
`setUserObject`

### addAnimationTrack

`public void` **`addAnimationTrack`**`(AnimationTrack animationTrack)`

Adds the given AnimationTrack to this Object3D, potentially changing the order and indices of the previously added tracks. The position at which the track is inserted among the existing tracks is deliberately left undefined. This gives implementations the freedom to select a data structure that best fits their needs, instead of mandating a particular kind of data structure.

The added animation track must be compatible with this Object3D. For example, to animate the diffuse color of a Material, the target property of the AnimationTrack must be `DIFFUSE_COLOR`. The target property is selected when constructing an AnimationTrack.

Multiple AnimationTracks can target the same property in the same object, in which case the value of the target property is a weighted linear combination of the individual tracks; see AnimationController for more information.

**Parameters:**
`animationTrack` - a compatible animation track to attach to this object

**Throws:**
`java.lang.NullPointerException` - if `animationTrack` is null
`java.lang.IllegalArgumentException` - if `animationTrack` is incompatible with this Object3D
`java.lang.IllegalArgumentException` - if `animationTrack` is already attached to this Object3D
`java.lang.IllegalArgumentException` - if `animationTrack` is targeting the same property of this Object3D as a previously added AnimationTrack, but does not have the same keyframe size

### getAnimationTrack

`public AnimationTrack` **`getAnimationTrack`**`(int index)`

Gets an AnimationTrack by index. Valid indices range from zero up to the value returned by `getAnimationTrackCount` minus one. Note that the index of any AnimationTrack may change whenever a track is added to or removed from this Object3D. See `addAnimationTrack` for more information.

**Parameters:**
`index` - index of the AnimationTrack to be retrieved

**Returns:**

the AnimationTrack at the given index

**Throws:**

`java.lang.IndexOutOfBoundsException` - if `index < 0 || index >= getAnimationTrackCount`

## removeAnimationTrack

```
public void removeAnimationTrack(AnimationTrack animationTrack)
```

Removes the given AnimationTrack from this Object3D, potentially changing the order and indices of the remaining tracks. If the given animation track is not associated with this object, or is null, the request to remove it is silently ignored.

**Parameters:**

`animationTrack` - the AnimationTrack to detach from this Object3D

## getAnimationTrackCount

```
public int getAnimationTrackCount()
```

Gets the number of AnimationTracks currently associated with this Object3D.

**Returns:**

the number of AnimationTracks bound to this Object3D

**javax.microedition.m3g**

## Class PolygonMode

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.PolygonMode
```

public class **PolygonMode**
extends Object3D

An Appearance component encapsulating polygon-level attributes. This includes settings related to back/front face culling, polygon winding, lighting computations, perspective correction, and shading.

*Winding* specifies which side of a polygon is the front face. Winding can be set to either clockwise (CW) or counter-clockwise (CCW). If the screen-space vertices of a polygon are in the order specified by the winding, then the polygon is front-facing. If the vertices are in the reverse order, then the polygon is back-facing.

*Culling* determines which side of a polygon is removed from processing prior to rasterization: the back face, the front face, or neither. Culling both faces is not allowed, as there are many other ways to make a piece of geometry invisible.

Lighting may operate in either one-sided or two-sided mode. In *one-sided* mode, a single color is computed for each vertex, based on the vertex normal, light source parameters, and material parameters. The same color is used in shading both the front face and the back face of the polygon. In *two-sided* mode, the colors for the back face of a polygon are computed separately and with reversed normals (**n**' = -**n**). Regardless of the lighting mode, the same set of Material parameters is used for both sides of the polygon. See the Material class description for more information on lighting.

There are two choices for polygon *shading*, smooth and flat. Smooth shading means that a color is computed separately for each pixel. This may be done by linear interpolation between vertex colors (also known as Gouraud shading), but implementations are also allowed to substitute a more accurate model. Flat shading means that the color computed for the *third* vertex of a triangle is used across the whole triangle.

If *local camera lighting* is disabled, the direction vector from the camera to the vertex being lit is approximated with (0 0 -1). If local camera lighting is enabled, the direction is computed based on the true camera position. This results in more accurate specular highlights. Note that local camera lighting only has an effect on the specular component of the lighting equation; the ambient and diffuse components remain unaffected. The local camera lighting flag is only a hint, so some implementations may not respect it. The application may use the `getProperties` method in Graphics3D to find out if the hint is supported.

*Perspective correction* is a generic term for techniques that eliminate artifacts caused by the screen-space interpolation of texture coordinates, colors and fog. The lack of perspective correction is especially evident on large textured polygons: the texture is distorted and seems to "crawl" on the surface as the viewing angle changes.

The perspective correction flag is only a hint, so some implementations may not respect it. Also, no particular method of implementing it is mandated or preferred. For example, some implementations may choose to do perspective correction for texture coordinates only. The application may use the `getProperties` method in Graphics3D to find out if the hint is supported.

**See Also:**
  Binary format

## Field Summary

| | |
|---|---|
| `static int` | **`CULL_BACK`**<br>A parameter to `setCulling`, specifying that the back-facing side of a polygon is not to be drawn. |
| `static int` | **`CULL_FRONT`**<br>A parameter to `setCulling`, specifying that the front-facing side of a polygon is not to be drawn. |
| `static int` | **`CULL_NONE`**<br>A parameter to `setCulling`, specifying that both faces of a polygon are to be drawn. |
| `static int` | **`SHADE_FLAT`**<br>A parameter to `setShading`, specifying that flat shading is to be used. |
| `static int` | **`SHADE_SMOOTH`**<br>A parameter to `setShading`, specifying that smooth shading is to be used. |
| `static int` | **`WINDING_CCW`**<br>A parameter to `setWinding`, specifying that a polygon having its vertices in counter-clockwise order in screen space is to be considered front-facing. |
| `static int` | **`WINDING_CW`**<br>A parameter to `setWinding`, specifying that a polygon having its vertices in clockwise order in screen space is to be considered front-facing. |

## Constructor Summary

| |
|---|
| **`PolygonMode`**`()`<br>Constructs a PolygonMode object with default values. |

## Method Summary

| | |
|---|---|
| `int` | **`getCulling`**`()`<br>Retrieves the current polygon culling mode. |
| `int` | **`getShading`**`()`<br>Retrieves the current polygon shading mode. |
| `int` | **`getWinding`**`()`<br>Retrieves the current polygon winding mode. |
| `boolean` | **`isLocalCameraLightingEnabled`**`()`<br>Queries whether local camera lighting is enabled. |
| `boolean` | **`isPerspectiveCorrectionEnabled`**`()`<br>Queries whether perspective correction is enabled. |
| `boolean` | **`isTwoSidedLightingEnabled`**`()`<br>Queries whether two-sided lighting is enabled. |
| `void` | **`setCulling`**`(int mode)`<br>Sets the polygon culling mode. |

| | |
|---|---|
| void | **setLocalCameraLightingEnable**(boolean enable)<br>Enables or disables local camera lighting. |
| void | **setPerspectiveCorrectionEnable**(boolean enable)<br>Enables or disables perspective correction. |
| void | **setShading**(int mode)<br>Sets the polygon shading mode. |
| void | **setTwoSidedLightingEnable**(boolean enable)<br>Enables or disables two-sided lighting. |
| void | **setWinding**(int mode)<br>Sets the polygon winding mode to clockwise or counter-clockwise. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject |

## Field Detail

### CULL_BACK

`public static final int CULL_BACK`

A parameter to `setCulling`, specifying that the back-facing side of a polygon is not to be drawn.

**See Also:**
Constant Field Values

### CULL_FRONT

`public static final int CULL_FRONT`

A parameter to `setCulling`, specifying that the front-facing side of a polygon is not to be drawn.

**See Also:**
Constant Field Values

### CULL_NONE

`public static final int CULL_NONE`

A parameter to `setCulling`, specifying that both faces of a polygon are to be drawn.

**See Also:**
Constant Field Values

### SHADE_FLAT

`public static final int` **`SHADE_FLAT`**

A parameter to `setShading`, specifying that flat shading is to be used.

**See Also:**
Constant Field Values

### SHADE_SMOOTH

`public static final int` **`SHADE_SMOOTH`**

A parameter to `setShading`, specifying that smooth shading is to be used.

**See Also:**
Constant Field Values

### WINDING_CCW

`public static final int` **`WINDING_CCW`**

A parameter to `setWinding`, specifying that a polygon having its vertices in counter-clockwise order in screen space is to be considered front-facing.

**See Also:**
Constant Field Values

### WINDING_CW

`public static final int` **`WINDING_CW`**

A parameter to `setWinding`, specifying that a polygon having its vertices in clockwise order in screen space is to be considered front-facing.

**See Also:**
Constant Field Values

## Constructor Detail

### PolygonMode

`public` **`PolygonMode`**`()`

Constructs a PolygonMode object with default values. The default values are as follows:

- culling : `CULL_BACK`
- winding : `WINDING_CCW`
- shading : `SHADE_SMOOTH`
- two-sided lighting: *false* (disabled)
- local camera lighting : *false* (disabled)
- perspective correction : *false* (disabled)

## Method Detail

### setCulling

`public void setCulling(int mode)`

Sets the polygon culling mode. The culling mode defines which sides of a polygon are culled (that is, not rendered). The winding mode, on the other hand, defines which side is considered to be the front. See the class description for more information.

**Parameters:**
`mode` - the culling mode to set: back, front or none
**Throws:**
`java.lang.IllegalArgumentException` - if `mode` is not one of `CULL_BACK, CULL_FRONT, CULL_NONE`
**See Also:**
`getCulling`

### getCulling

`public int getCulling()`

Retrieves the current polygon culling mode.

**Returns:**
the current culling mode; one of the symbolic constants
**See Also:**
`setCulling`

### setWinding

`public void setWinding(int mode)`

Sets the polygon winding mode to clockwise or counter-clockwise. The winding mode defines which side of a polygon is considered to be the front. This and the culling mode together determine which sides of a polygon are rendered. The winding mode has consequences on lighting, as well, if two-sided lighting is enabled. See the class description for more information.

**Parameters:**

`mode` - the winding mode to set: clockwise or counter-clockwise

**Throws:**

`java.lang.IllegalArgumentException` - if `mode` is not one of `WINDING_CCW`, `WINDING_CW`

**See Also:**

`getWinding`

## getWinding

```
public int getWinding()
```

Retrieves the current polygon winding mode.

**Returns:**

the current winding mode; one of the symbolic constants

**See Also:**

`setWinding`

## setShading

```
public void setShading(int mode)
```

Sets the polygon shading mode. The shading mode defines whether a single color is assigned to the whole polygon (flat shading) or if a color is computed separately for each pixel (smooth shading). See the class description for more information.

**Parameters:**

`mode` - the shading mode to set: flat or smooth

**Throws:**

`java.lang.IllegalArgumentException` - if `mode` is not one of `SHADE_FLAT`, `SHADE_SMOOTH`

**See Also:**

`getShading`

## getShading

```
public int getShading()
```

Retrieves the current polygon shading mode.

**Returns:**

the current shading mode: flat or smooth

**See Also:**

`setShading`

## setTwoSidedLightingEnable

```
public void setTwoSidedLightingEnable(boolean enable)
```

Enables or disables two-sided lighting. If two-sided lighting is enabled, the lit colors for the front and back faces of a polygon are computed differently. Otherwise, both faces are assigned the same color. See the class description for more information.

**Parameters:**

`enable` - *true* to enable two-sided lighting; *false* to use one-sided lighting

**See Also:**

`isTwoSidedLightingEnabled`

### isTwoSidedLightingEnabled

`public boolean isTwoSidedLightingEnabled()`

Queries whether two-sided lighting is enabled.

**Returns:**

*true* if two-sided lighting is enabled; *false* if not

**See Also:**

`setTwoSidedLightingEnable`

### setLocalCameraLightingEnable

`public void setLocalCameraLightingEnable(boolean enable)`

Enables or disables local camera lighting. Note that this is only a hint: the implementation may or may not obey it. See the class description for further discussion on local camera lighting.

**Parameters:**

`enable` - *true* to enable local camera lighting; *false* to disable it

**See Also:**

`isLocalCameraLightingEnabled`

### isLocalCameraLightingEnabled

`public boolean isLocalCameraLightingEnabled()`

Queries whether local camera lighting is enabled. Note that the set value is returned, regardless of whether the implementation obeys it or not.

**Returns:**

*true* if local camera lighting is enabled; *false* if not

**Since:**

M3G 1.1

**See Also:**

`setLocalCameraLightingEnable`

### setPerspectiveCorrectionEnable

`public void setPerspectiveCorrectionEnable(boolean enable)`

Enables or disables perspective correction. Note that this is only a hint: the implementation may or may not obey it. See the class description for further discussion on perspective correction.

**Parameters:**

`enable` - *true* to enable perspective correction; *false* to disable it

**See Also:**

`isPerspectiveCorrectionEnabled`

## isPerspectiveCorrectionEnabled

public boolean **isPerspectiveCorrectionEnabled**()

Queries whether perspective correction is enabled. Note that the set value is returned, regardless of whether the implementation obeys it or not.

**Returns:**

*true* if perspective correction is enabled; *false* if not

**Since:**

M3G 1.1

**See Also:**

`setPerspectiveCorrectionEnable`

**javax.microedition.m3g**

# Class RayIntersection

```
java.lang.Object
  └─javax.microedition.m3g.RayIntersection
```

public class **RayIntersection**
extends java.lang.Object

A RayIntersection object is filled in by the `pick` methods in Group. RayIntersection stores a reference to the intersected Mesh or Sprite3D and information about the intersection point. RayIntersection is strictly a run-time object; it cannot be loaded from a file by Loader.

| **Constructor Summary** |
|---|
| `RayIntersection()`<br>Constructs a new RayIntersection object with default values. |

| **Method Summary** | |
|---|---|
| `float` | `getDistance()`<br>Retrieves the distance from the pick ray origin to the intersection point. |
| `Node` | `getIntersected()`<br>Retrieves the picked Mesh or Sprite3D object. |
| `float` | `getNormalX()`<br>Retrieves the X component of the surface normal at the intersection point. |
| `float` | `getNormalY()`<br>Retrieves the Y component of the surface normal at the intersection point. |
| `float` | `getNormalZ()`<br>Retrieves the Z component of the surface normal at the intersection point. |
| `void` | `getRay(float[] ray)`<br>Retrieves the origin (ox oy oz) and direction (dx dy dz) of the pick ray, in that order. |
| `int` | `getSubmeshIndex()`<br>Retrieves the index of the submesh where the intersection point is located within the intersected Mesh. |
| `float` | `getTextureS(int index)`<br>Retrieves the S texture coordinate at the intersection point on the picked Mesh or Sprite3D. |
| `float` | `getTextureT(int index)`<br>Retrieves the T texture coordinate at the intersection point on the picked Mesh or Sprite3D. |

## Constructor Detail

### RayIntersection

`public **RayIntersection**()`

Constructs a new RayIntersection object with default values. The default values are as follows:

- ray origin : (0 0 0)
- ray direction : (0 0 1)
- intersected node : null
- intersected submesh index : 0
- distance to intersection point : 0.0
- all texture coordinates : 0.0
- normal vector: (0 0 1)

## Method Detail

### getIntersected

`public Node **getIntersected**()`

Retrieves the picked Mesh or Sprite3D object. Other types of Nodes are not pickable and hence can not be returned by this method.

**Returns:**
the picked Mesh or Sprite3D object

### getRay

`public void **getRay**(float[] ray)`

Retrieves the origin (ox oy oz) and direction (dx dy dz) of the pick ray, in that order. The ray origin and direction vector are specified in the coordinate system of the Group node where the `pick` method was called from. The returned direction vector is the same that is used to compute the distance measure from the pick point to the object in `getDistance`.

Note that if the application itself provides the pick ray origin and direction to the `pick` method, this method simply returns the same information; in particular, the direction vector is returned as is, not normalized. On the other hand, if the application uses the other `pick` method, where only a Camera and a point on the viewing plane are specified, the ray origin and direction would not otherwise be readily available.

This method together with `getDistance` enables the point of intersection to be computed conveniently, as shown in the example below.

**Parameters:**
`ray` - a float array to fill in with the pick ray origin and direction, in that order
**Throws:**
`java.lang.NullPointerException` - if `ray` is null
`java.lang.IllegalArgumentException` - if `ray.length < 6`
**Example:**
**Computing the ray intersection point.**

```
float x, y, z;                  // the intersection point
 float [] ray = new float[6];    // ray origin and direction
 RayIntersection ri = new RayIntersection();

 // Pick through the center of the viewport

 if (myGroup.pick(-1, 0.5f, 0.5f, myCamera, ri) == true)
 {
     ri.getRay(ray);
     x = ray[0] + ray[3] * ri.getDistance();
     y = ray[1] + ray[4] * ri.getDistance();
     z = ray[2] + ray[5] * ri.getDistance();
 }
```

## getDistance

`public float` **`getDistance`**`()`

Retrieves the distance from the pick ray origin to the intersection point. The distance is normalized to the length of the given pick ray (1.0 = ray length). The length of the pick ray is defined as sqrt($dx^2$ + $dy^2$ + $dz^2$), where (dx dy dz) is the direction vector of the ray. The direction vector itself can be obtained using `getRay`.

The normalized distance is convenient, because it is independent of the transformations of the intersected Node and its ancestors, including any non-uniform scales and other non-length preserving transformations. The distance to the intersection point can be used for simple collision detection, for instance.

**Returns:**
normalized distance from the pick ray origin to the intersection point

## getSubmeshIndex

`public int` **`getSubmeshIndex`**`()`

Retrieves the index of the submesh where the intersection point is located within the intersected Mesh. This allows the application to identify, for example, the texture image that is displayed at the intersection point. The submesh index is only applicable to Meshes; its value is always set to zero if the picked object is a Sprite3D.

**Returns:**
index of the intersected submesh (always 0 for sprites)

## getTextureS

`public float` **`getTextureS`**`(int index)`

Retrieves the S texture coordinate at the intersection point on the picked Mesh or Sprite3D. For meshes, there can be between zero and N texture coordinates, where N is the number of texturing units supported by the implementation. If a texturing unit is disabled, the corresponding texture coordinates are undefined. For sprites, there is always exactly one pair of valid texture coordinates (at index zero); the rest of the coordinates are undefined.

If the picked object is a Mesh, the returned coordinates represent the texture coordinates *after* applying the texture transformation and projection, but *before* possible clamping. In the case of a Sprite3D, the returned coordinates are always between [0, 1], where (0, 0) is the upper left corner of the sprite image. Note that the sprite crop rectangle has no effect on the returned values.

**Parameters:**
`index` - index of the texturing unit to get the texture coordinate of

**Returns:**
the S texcoord of the specified texturing unit at the intersection point

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `index != [0, N]` where `N` is the implementation specific maximum texturing unit index

## getTextureT

`public float` **`getTextureT`**`(int index)`

Retrieves the T texture coordinate at the intersection point on the picked Mesh or Sprite3D. See `getTextureS` for more information.

**Parameters:**
`index` - index of the texturing unit to get the texture coordinate of

**Returns:**
the T texcoord of the specified texturing unit at the intersection point

**Throws:**
`java.lang.IndexOutOfBoundsException` - if `index != [0, N]` where `N` is the implementation specific maximum texturing unit index

## getNormalX

`public float` **`getNormalX`**`()`

Retrieves the X component of the surface normal at the intersection point. The normal is specified in the coordinate system of the intersected Node, and is always unit length. If the picked object is a Sprite3D, the normal vector is always (0 0 1). If the object is a Mesh with no vertex normals, the returned normal is undefined.

**Returns:**
the X component of the surface normal at the intersection point

## getNormalY

`public float` **`getNormalY`**`()`

Retrieves the Y component of the surface normal at the intersection point. See `getNormalX` for more information.

**Returns:**
the Y component of the surface normal at the intersection point

## getNormalZ

## `public float **getNormalZ**()`

Retrieves the Z component of the surface normal at the intersection point. See `getNormalX` for more information.

**Returns:**

the Z component of the surface normal at the intersection point

**javax.microedition.m3g**

## Class SkinnedMesh

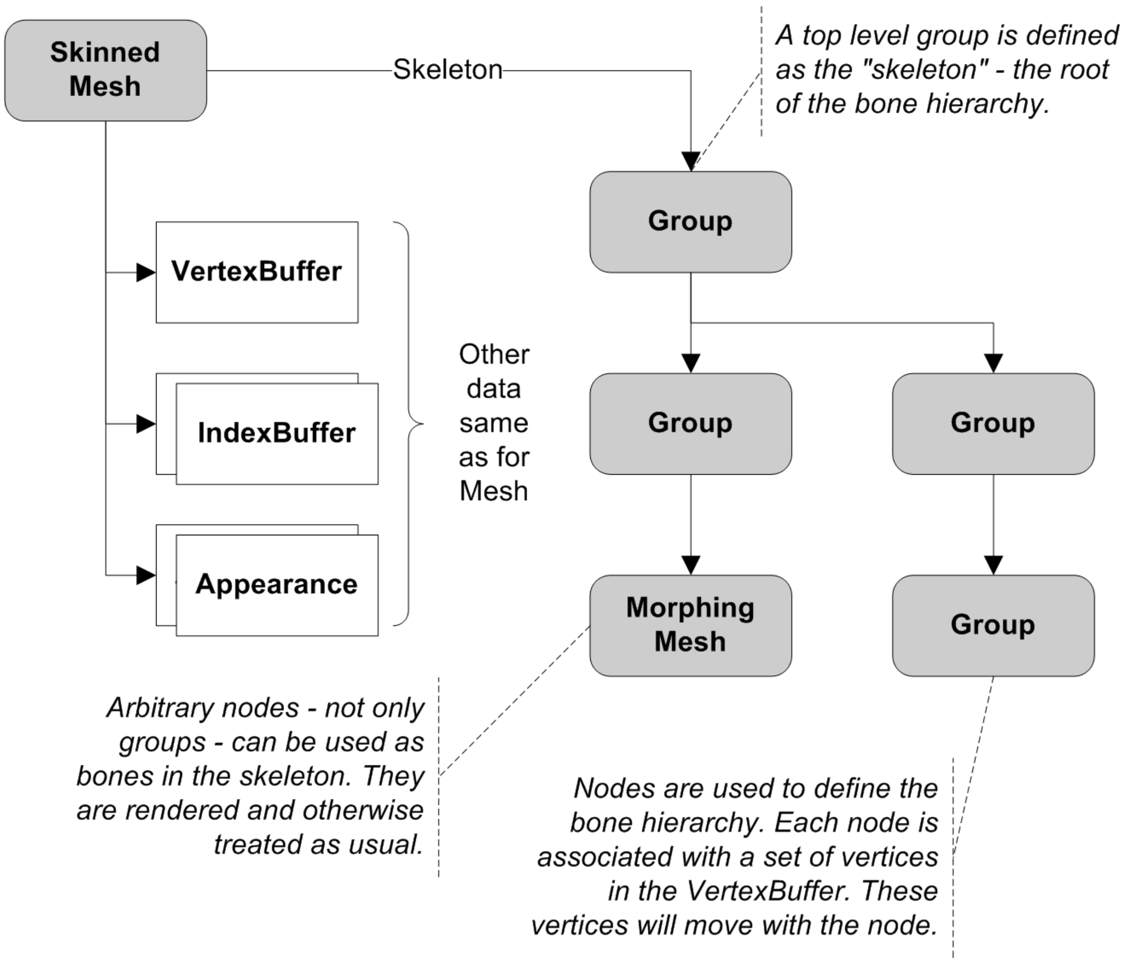
```
java.lang.Object
  └─javax.microedition.m3g.Object3D
      └─javax.microedition.m3g.Transformable
          └─javax.microedition.m3g.Node
              └─javax.microedition.m3g.Mesh
                  └─javax.microedition.m3g.SkinnedMesh
```

public class **SkinnedMesh**
extends Mesh

A scene graph node that represents a skeletally animated polygon mesh.

Vertex positions in a SkinnedMesh can be associated with multiple separately transforming Nodes, with a weight factor specified for each. This enables groups of vertices to transform independently of each other while smoothly deforming the polygon mesh "skin" with the vertices. This style of animation is highly efficient for animated characters.

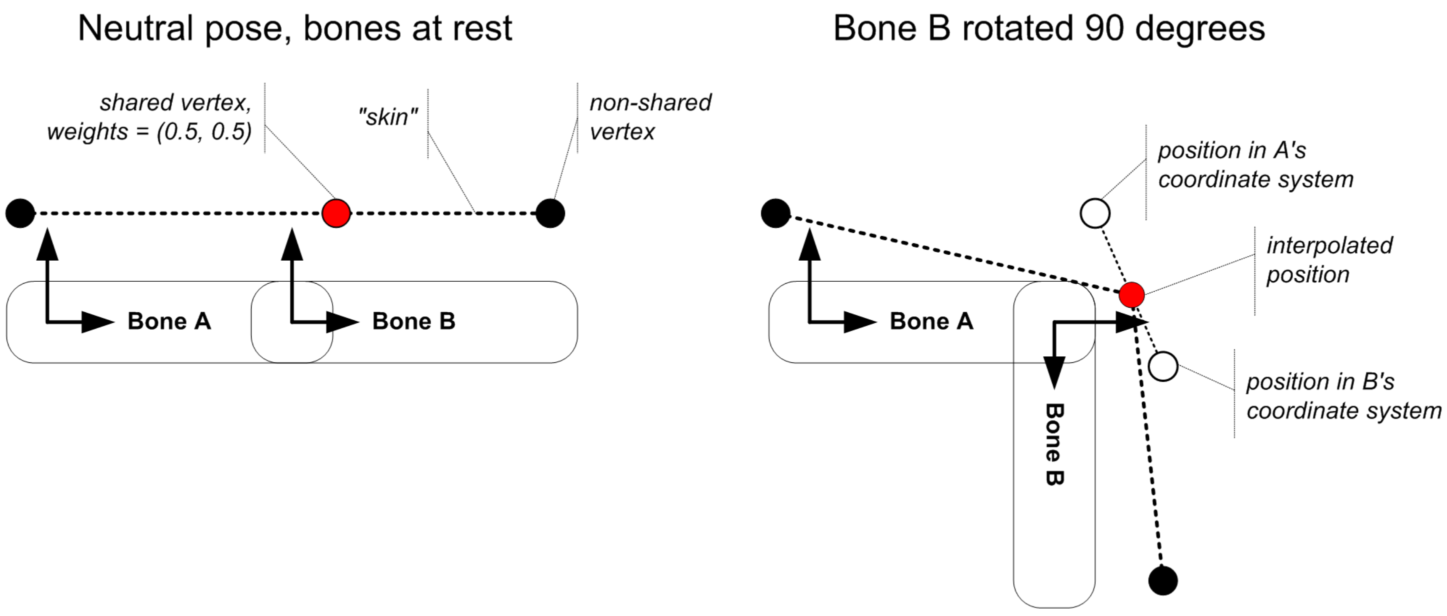
The structure of a SkinnedMesh is shown in the figure below.


Skinned Mesh
Skeleton
A top level group is defined as the "skeleton" - the root of the bone hierarchy.
Group
VertexBuffer
IndexBuffer
Appearance
Other data same as for Mesh
Group
Group
Morphing Mesh
Group
Arbitrary nodes - not only groups - can be used as bones in the skeleton. They are rendered and otherwise treated as usual.
Nodes are used to define the bone hierarchy. Each node is associated with a set of vertices in the VertexBuffer. These vertices will move with the node.

A SkinnedMesh node is the parent of its skeleton group, and vice versa, the skeleton is the only child of the SkinnedMesh. In other words, `this.getSkeleton().getParent() == this`. The skeleton group and its descendants constitute a branch in the scene graph, and that branch is traversed just like any other branch during rendering and picking. Any sprites and meshes, including skinned meshes, contained in the skeleton group are therefore rendered as usual. This allows, for example, a game character to have a weapon in its hand, such that the weapon is a separate node that can be easily interchanged with another.

## Vertex transformation

Each vertex is transformed once for each Node affecting it. The results are then blended together according to the weight factors of each node. To get an initial idea of how this works, see the figure below. A more formal definition follows.


Neutral pose, bones at rest
shared vertex, weights = (0.5, 0.5)
"skin"
non-shared vertex
Bone A
Bone B
Bone B rotated 90 degrees
position in A's coordinate system
interpolated position
position in B's coordinate system
Bone A
Bone B

Let us denote the set of nodes (bones) associated with a vertex by { $N_1$, $N_2$, ..., $N_N$ }. Let us also denote by $\mathbf{M}_i$ the transformation *from* the local coordinate system of node $N_i$ *to* a *reference coordinate system*. The choice of the reference coordinate system is not critical; depending on the implementation, good choices may include the world coordinate system, the coordinate system of the SkinnedMesh node, or the coordinate system of the current camera. Finally, let us denote the weight associated with node $N_i$ as $W_i$. The blended position of a vertex in the reference coordinate system is then:

$$\mathbf{v}' = \text{sum} [\, w_i \mathbf{M}_i \mathbf{B}_i \mathbf{v}]$$

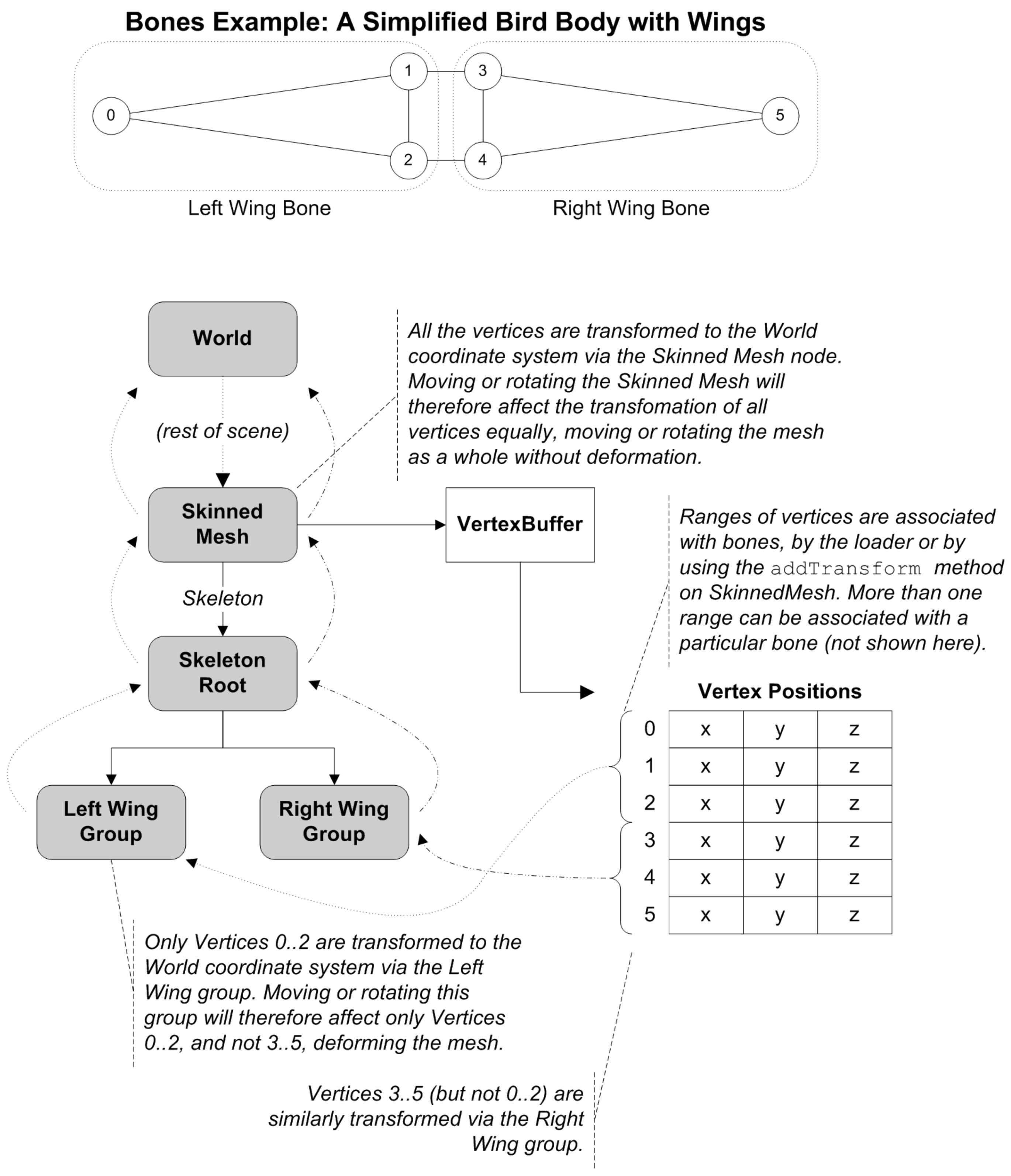
where

- 0 <= i < N, where N is the number of bones associated with **v**;
- **v** is the original vertex position in the source VertexBuffer;
- $\mathbf{B}_i$ is the "at rest" transformation from the SkinnedMesh node to bone $N_i$;
- $\mathbf{M}_i$ is the transformation from bone $N_i$ to the chosen reference coordinate system (e.g. world coordinates);
- $w_i$ is the normalized weight of bone $N_i$, computed as $w_i = W_i / (W_1 + ... + W_N)$.

Finally, the blended vertex position **v**' is transformed from the chosen reference coordinate system to the camera space as usual. Note that when computing the normalized weights $w_i$, 0 / 0 = 0.

If a vertex **v** has no transformations associated with it, as is the case for all vertices in a newly constructed SkinnedMesh, the vertex lies implicitly in the coordinate system of the SkinnedMesh node itself. That is, a SkinnedMesh in its initial state is equivalent to an ordinary Mesh. When a vertex is explicitly associated with any bone in the skeleton, the implicit association with the SkinnedMesh node is removed.

The transformation of vertices is illustrated in the figure below.



*All the vertices are transformed to the World coordinate system via the Skinned Mesh node. Moving or rotating the Skinned Mesh will therefore affect the transfomation of all vertices equally, moving or rotating the mesh as a whole without deformation.*

*Ranges of vertices are associated with bones, by the loader or by using the* `addTransform` *method on SkinnedMesh. More than one range can be associated with a particular bone (not shown here).*

|   |   |   |   |
|---|---|---|---|
| 0 | x | y | z |
| 1 | x | y | z |
| 2 | x | y | z |
| 3 | x | y | z |
| 4 | x | y | z |
| 5 | x | y | z |

*Only Vertices 0..2 are transformed to the World coordinate system via the Left Wing group. Moving or rotating this group will therefore affect only Vertices 0..2, and not 3..5, deforming the mesh.*

*Vertices 3..5 (but not 0..2) are similarly transformed via the Right Wing group.*

## Deferred exceptions

Any special cases and exceptions that are defined for Mesh also apply for SkinnedMesh. An extra exception case is introduced due to the vertex indices set by `addTransform`. If any part of the skinned mesh is needed for picking or rendering, then every bone in that mesh must refer to a valid range of vertex indices, otherwise an IllegalStateException will be thrown. The indices cannot be validated until when they are actually needed, that is, when rendering or picking. This is because the application may change the length of the associated VertexBuffer, and consequently make the indices invalid or valid, at any time.

**See Also:**

Binary format

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| `SkinnedMesh(VertexBuffer vertices, IndexBuffer[] submeshes, Appearance[] appearances, Group skeleton)` Constructs a new SkinnedMesh with the given vertices, submeshes and skeleton. |
| `SkinnedMesh(VertexBuffer vertices, IndexBuffer submesh, Appearance appearance, Group skeleton)` Constructs a new SkinnedMesh with the given vertices, submesh and skeleton. |

## Method Summary

| | |
|---|---|
| `void` | `addTransform(Node bone, int weight, int firstVertex, int numVertices)` Associates a weighted transformation, or "bone", with a range of vertices in this SkinnedMesh. |
| `void` | `getBoneTransform(Node bone, Transform transform)` Returns the at-rest transformation for a bone node. |
| `int` | `getBoneVertices(Node bone, int[] indices, float[] weights)` Returns the number of vertices influenced by the given bone, filling in the vertices and their weights to the given arrays. |
| `Group` | `getSkeleton()` Returns the skeleton Group of this SkinnedMesh. |

| **Methods inherited from class javax.microedition.m3g.Mesh** |
|---|
| `getAppearance, getIndexBuffer, getSubmeshCount, getVertexBuffer, setAppearance` |

| **Methods inherited from class javax.microedition.m3g.Node** |
|---|
| `align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope` |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### SkinnedMesh

```
public SkinnedMesh(VertexBuffer vertices,
                   IndexBuffer submesh,
                   Appearance appearance,
                   Group skeleton)
```

Constructs a new SkinnedMesh with the given vertices, submesh and skeleton. Except for the skeleton, the behavior of this constructor is identical to the corresponding constructor in Mesh; refer to that for more information.

No transformations are initially associated with the vertices. The behavior of a newly constructed SkinnedMesh is therefore equivalent to an ordinary Mesh.

**Parameters:**

`vertices` - a VertexBuffer to use for this mesh
`submesh` - an IndexBuffer defining the triangle strips to draw
`appearance` - an Appearance to use for this mesh, or null
`skeleton` - a Group containing the skeleton of this SkinnedMesh

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `submesh` is null
`java.lang.NullPointerException` - if `skeleton` is null
`java.lang.IllegalArgumentException` - if `skeleton` is a World node
`java.lang.IllegalArgumentException` - if `skeleton` already has a parent

### SkinnedMesh

```
public SkinnedMesh(VertexBuffer vertices,
                   IndexBuffer[] submeshes,
                   Appearance[] appearances,
                   Group skeleton)
```

Constructs a new SkinnedMesh with the given vertices, submeshes and skeleton. Except for the skeleton, the behavior of this constructor is identical to the corresponding constructor in Mesh; refer to that for more information.

No transformations are initially associated with the vertices. The behavior of a newly constructed SkinnedMesh is therefore equivalent to an ordinary Mesh.

**Parameters:**

`vertices` - a VertexBuffer to use for all submeshes in this mesh
`submeshes` - an IndexBuffer array defining the submeshes to draw
`appearances` - an Appearance array parallel to `submeshes`, or null
`skeleton` - a Group containing the skeleton of this SkinnedMesh

**Throws:**

`java.lang.NullPointerException` - if `vertices` is null
`java.lang.NullPointerException` - if `submeshes` is null
`java.lang.NullPointerException` - if any element in `submeshes` is null
`java.lang.NullPointerException` - if `skeleton` is null
`java.lang.IllegalArgumentException` - if `submeshes` is empty
`java.lang.IllegalArgumentException` - if `(appearances != null) && (appearances.length < submeshes.length)`
`java.lang.IllegalArgumentException` - if `skeleton` is a World node
`java.lang.IllegalArgumentException` - if `skeleton` already has a parent

## Method Detail

### getSkeleton

```
public Group getSkeleton()
```

Returns the skeleton Group of this SkinnedMesh. The skeleton group is set in the constructor and can not be removed or replaced with another Group afterwards. All transform reference nodes (bones) must be descendants of the skeleton group; this is enforced by `addTransform`.

**Returns:**

the skeleton Group of this SkinnedMesh

### addTransform

```
public void addTransform(Node bone,
                         int weight,
                         int firstVertex,
                         int numVertices)
```

Associates a weighted transformation, or "bone", with a range of vertices in this SkinnedMesh. See the transformation equation in the class description for how the transformations are applied on vertices.

An integer weight is supplied as a parameter for each added transformation. Prior to solving the transformation equation, the weights are automatically normalized on a per-vertex basis such that the individual weights are between [0, 1] and their sum is 1.0. This is done by dividing each weight pertaining to a vertex by the sum of all weights pertaining to that vertex. For example, if two bones with any equal weights overlap on a vertex, each bone will get a final weight of 0.5.

Automatic normalization of weights is convenient because it significantly reduces the number of times that this method must be called (and hence the amount of data that must be stored and transmitted) in cases where more than one bone is typically associated with each vertex.

The same Node may appear multiple times among the bones. This is to allow multiple disjoint sets of vertices to be attached to the same bone.

The number of bones that can be associated with a single vertex is unlimited, except for the amount of available memory. However, there is an implementation defined limit (N) to the number of bones that can actually have an effect on any single vertex. If more than N bones are active on a vertex, the implementation is required to select the N bones with highest weights. In case of a tie (multiple bones with equal weights competing for the last slot), the selection method is undefined but must be deterministic. The limit N can be queried from `getProperties`.

The "at-rest" transformation from this SkinnedMesh to the given bone is set equal to `this.getTransformTo(bone)`. If the at-rest transformation cannot be computed, an ArithmeticException is thrown; see `Node.getTransformTo` for more information. If `addTransform` is called more than once for the same bone, the final at-rest transformation of that bone can become any of the at-rest transformations that were in effect during those calls.

**Parameters:**

`bone` - a node in the skeleton group to transform the vertices with
`weight` - weight of `bone`; any positive integer is accepted
`firstVertex` - index of the first vertex to be affected by `bone`
`numVertices` - number of consecutive vertices to attach to the bone node

**Throws:**

`java.lang.NullPointerException` - if `bone` is null
`java.lang.IllegalArgumentException` - if `bone` is not the skeleton Group or one of its descendants
`java.lang.IllegalArgumentException` - if `weight <= 0`
`java.lang.IllegalArgumentException` - if `numVertices <= 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex + numVertices > 65535`
`java.lang.ArithmeticException` - if the at-rest transformation cannot be computed

## getBoneTransform

```
public void getBoneTransform(Node bone,
                             Transform transform)
```

Returns the at-rest transformation for a bone node. This is the transformation stored in `addTransform` as described in the documentation there.

If the given node is in the skeleton group of this Mesh, but has no vertices associated with it according to `getBoneVertices`, the returned transformation is undefined.

**Parameters:**

`bone` - the bone node
`transform` - the Transform object to receive the bone transformation

**Throws:**

`java.lang.NullPointerException` - if `bone` is null
`java.lang.NullPointerException` - if `transform` is null
`java.lang.IllegalArgumentException` - if `bone` is not in the skeleton group of this mesh

**Since:**

M3G 1.1

**See Also:**

`getBoneVertices`, `addTransform`

## getBoneVertices

```
public int getBoneVertices(Node bone,
                           int[] indices,
                           float[] weights)
```

Returns the number of vertices influenced by the given bone, filling in the vertices and their weights to the given arrays. If either or both of the arrays are null, only the number of vertices is returned.

Each bone node may be associated with disjoint sets of vertices via multiple `addTransform` calls. The vertices are therefore returned as explicit vertex indices with corresponding per-vertex bone weights. The order of the returned index-weight pairs is implementation-dependent. Duplicate indices and indices with zero weight are not returned. The returned weights are normalized so that all weights (from all contributing bones) corresponding to a single vertex sum to one.

Implementations are only required to associate with a vertex the N bones with the highest weights, where N is the maximum number of transformations per vertex as queried from `Graphics3D.getProperties`. For the other bones, a weight of zero can be assumed. The returned weights for each vertex must still sum to one.

The minimum precision requirements for vertex weights are less than the general requirements given in the package description. The returned weights must have a precision equivalent to a minimum internal precision of 8-bit fixed point.

**Parameters:**

`bone` - the bone node
`indices` - an array to store the vertex indices, or null to only query the number of vertices that will be returned
`weights` - an array to store the vertex weights, or null to only query the number of vertices that will be returned

**Returns:**

the number of vertices influenced by `bone`

**Throws:**

`java.lang.NullPointerException` - if `bone` is null
`java.lang.IllegalArgumentException` - if `bone` is not in the skeleton group of this mesh
`java.lang.IllegalArgumentException` - if neither of `indices` and `weights` is null, and the length of either is less than the number of vertices queried

**Since:**

M3G 1.1

**See Also:**

`addTransform`, `getBoneTransform`

**javax.microedition.m3g**

# Class Sprite3D

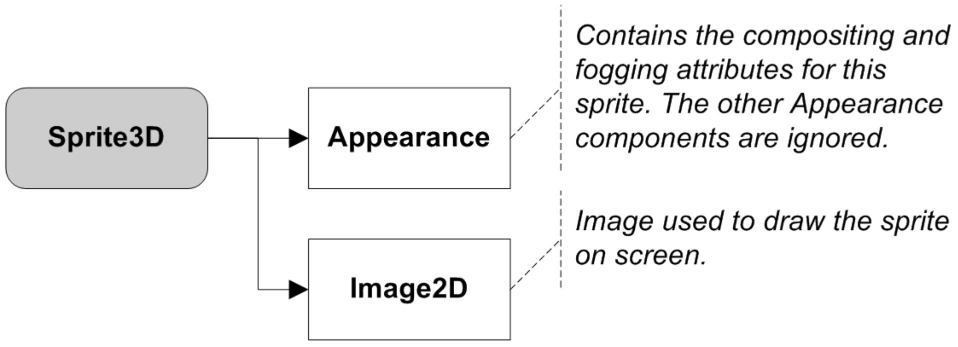
```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
          └ javax.microedition.m3g.Node
              └ javax.microedition.m3g.Sprite3D
```

public class **Sprite3D**
extends Node

A scene graph node that represents a 2-dimensional image with a 3D position.

Sprite3D is a fast, but functionally restricted alternative to textured geometry. A Sprite3D is rendered as a screen-aligned rectangular array of pixels with a constant depth. The apparent size of a sprite may be specified directly in pixels (an unscaled sprite) or indirectly using the transformation from the Sprite3D node to the camera space (a scaled sprite).

The structure of a Sprite3D object is shown in the figure below.


Sprite3D
Appearance
Image2D
Contains the compositing and fogging attributes for this sprite. The other Appearance components are ignored.
Image used to draw the sprite on screen.

## Sprite image data

The sprite image is stored as a reference to an Image2D. The image may be in any of the formats defined in Image2D. The width and height of the image are not limited in any way; in particular, they need not be powers of two. However, there is an implementation defined maximum size for the crop rectangle (the area of the sprite that is actually displayed). This can be queried with `getProperties`.

The displayed sprite image can be mirrored with respect to the X and/or Y axes by specifying a crop rectangle with a negative width and/or height, respectively.

If the referenced Image2D is mutable and is modified while it is bound to a Sprite3D, or a new Image2D is bound as the sprite image, the modifications are immediately reflected in the Sprite3D. Be aware, however, that changing or updating the sprite image may trigger time-consuming operations.

## Sprite positioning and scaling

The position of a sprite on screen is determined by projecting the origin of its coordinate system to screen coordinates.

The resulting 2D position is used as the center of the displayed pixel array. If this causes any part of the sprite to be placed off screen, then the sprite is clipped to the visible portion of the viewport as usual (refer to Graphics3D).

The depth value of a sprite is constant across the image, and is the depth of the origin of the Sprite3D coordinate system.

The width and height of an *unscaled* sprite on screen are measured in pixels, and they are equal to the (absolute) width and height of the crop rectangle. Recall that the crop rectangle dimensions may be negative to flip the pixels; this has no effect on the size of the sprite.

The width and height of a *scaled* sprite on screen are computed basically as if the sprite were a rectangle with unit-length sides, lying on the XY plane of its local coordinate system and centered at its origin. The contents of the crop rectangle are scaled to fill the projected unit rectangle. See the Implementation guidelines below for the details.

Because a sprite is always displayed as a screen-aligned rectangle, the effects of other than rigid-body transformations on sprites may not be immediately intuitive, even though they are well-defined. It is advised that, for example, non-uniform scaling and skewing be avoided in sprite modelview matrices. Similarly, oblique projections should be used with caution when sprites are present in the displayed scene.

## Sprite rendering attributes

The rendering attributes for a Sprite3D are determined by its Appearance, as is the case with Mesh objects. There are a number of properties in Appearance, however, that do not have a meaningful interpretation in this context. Thus, only the CompositingMode and Fog components and the layer index are taken into account when rendering a sprite. The rest of the components are ignored. This implies, in particular, that lighting does not apply for sprites.

## Sprite picking

Only scaled sprites can be picked. This is because the dimensions of an unscaled sprite are only defined in screen space, that is, after the viewport transformation. Since the viewport parameters are not available to the pick method, the dimensions of an unscaled sprite can not be computed.

Picking of scaled sprites is analogous to how they are rendered. Picking is done in normalized device coordinates (after projection, before viewport transformation), where the position, depth value and dimensions of scaled sprites are well defined. See the Implementation guidelines for how to calculate these.

Since the sprite size calculation requires a Camera, sprites are only pickable through the viewing plane, not from an arbitrary position in the scene. That is, of the two `pick` variants in Group, only the one that takes in a Camera as a parameter can be used in sprite picking. The other variant simply ignores all sprite nodes.

If a sprite is intersected by the pick ray, the pixel in the sprite image at the intersection point will be further tested for transparency. If the pixel is fully transparent, the sprite is not picked and the pick ray will continue towards objects that are further away. If the pixel is not fully transparent, the sprite is picked. A pixel is defined to be fully transparent if and only if it fails the alpha test (see `CompositingMode.setAlphaThreshold`) prior to applying the alpha factor.

## Implementation guidelines

Sprites do not provide any functionality that would not be available otherwise; they can always be substituted with textured meshes. However, the existence of the sprite primitive acts as a signal to the renderer that a very specific subset of functionality is required. This allows the rendering pipeline to avoid the overhead of transforming and lighting full geometry. It also allows the rasterizer to select an optimized drawing routine to place the pixels on the screen, without the potentially complex interpolation of parameters across the rectangle. This can be used to make sprites very much faster than textured meshes (especially in software) which in turn increases the richness of content that can be offered at the low end.

Filtering of scaled sprites can be implemented with the simple nearest neighbor algorithm, but implementations are recommended to apply a more sophisticated scheme, such as bilinear filtering with mipmapping. No application control over the filtering behavior is provided, however; if that is required, the application should use textured rectangles instead. For unscaled sprites, the implementation should ensure that rounding errors or similar do not yield unwelcome artifacts, such as pixel columns appearing and disappearing depending on the screen position of the sprite. This is particularly important for text labels.

### Implementing with textures

Sprite3D can be implemented with textured rectangles. To facilitate that, implementations are allowed to upscale or downscale the sprite image to power-of-two dimensions, preferably using bilinear filtering or better. Images exceeding the maximum texture size may be downscaled to the maximum size. However, implementations striving for best image quality would keep the large original image around, texturing the rectangle with the crop rectangle contents only. Note that if the sprite image is mutable, the original image must be kept around in any case.

The sprite alpha factor can be trivially implemented with the `MODULATE` texture blending mode, by setting the fragment alpha equal to the effective alpha factor and the fragment color components to 1.0. Another option is to premultiply the alpha factor into the alpha channel of the sprite image; however, the implementation must then make sure that the original alpha values can always be recovered intact.

### Computing the position and size

The position of a sprite on the viewport is simply the projected location of the Sprite3D node's origin. Similarly, the depth of the sprite is the projected depth of the origin.

The size of a sprite, in pixels, depends on whether the sprite is scaled or not. An unscaled sprite is the same size as its crop rectangle. Calculating the size of a scaled sprite is slightly more complicated. In principle, it only involves projecting the Sprite3D node's X and Y axes into screen space and computing their length, but to make the calculation well defined under arbitrary transformations, a few additional steps are required. The exact formula is given below.

Let us define **M** and **P** as the current modelview and projection matrices. The modelview matrix **M** is the concatenated transformation from the Sprite3D node into camera space (taking into account all of the transformation components of the Sprite3D, including the user-settable static matrix), and **P** is the projection matrix of the current camera.

Let us first transform the origin and two reference points, corresponding to the X and Y axes, from the Sprite3D node's coordinate system into camera space:

$$\mathbf{o}' = \mathbf{M}(0, 0, 0, 1)^T$$
$$\mathbf{x}' = \mathbf{M}(1, 0, 0, 1)^T$$
$$\mathbf{y}' = \mathbf{M}(0, 1, 0, 1)^T$$

We then compute the distances from the origin to the X and Y reference points. If the bottom row of the modelview matrix is not (0 0 0 1), the transformed points may have W values different from each other. The points are thus brought into an equal scale (W = 1) before computing the distance.

$$dx = |\,\mathbf{x}'/x'_w - \mathbf{o}'/o'_w|$$
$$dy = |\,\mathbf{y}'/y'_w - \mathbf{o}'/o'_w|$$

Note that the W components of the homogeneous points cancel out in the subtraction, and the lengths computed are those of the 3D vectors. We then define (new) X and Y reference points that lie on the X and Y axes of the camera and whose distances from the origin are dx and dy, respectively. Applying the projection matrix, we transform the origin and the reference points into clip space:

$$\mathbf{o}'' = \mathbf{P}\mathbf{o}'$$
$$\mathbf{x}'' = \mathbf{P}[\mathbf{o}' + (dx, 0, 0, 0)^T]$$
$$\mathbf{y}'' = \mathbf{P}[\mathbf{o}' + (0, dy, 0, 0)^T]$$

Again, we compute the distances from the origin to the X and Y reference points. This gives us the dimensions of the sprite in normalized device coordinates.

$$sx = | \mathbf{x}''/x''_w - \mathbf{o}''/o''_w|$$
$$sy = | \mathbf{y}''/y''_w - \mathbf{o}''/o''_w|$$

These dimensions are used when testing the sprite for an intersection with a pick ray. For rendering the sprite, we need to apply the viewport transformation (see Graphics3D) to obtain the final on-screen size in pixels:

$$w = 0.5\ sx\ w_{vp}$$
$$h = 0.5\ sy\ h_{vp}$$

where $w_{vp}$ and $h_{vp}$ are the dimensions of the viewport.

The pixels within the crop rectangle are mapped onto the resulting rectangle. If both the crop width and height are positive, then the top left pixel of the crop rectangle maps to the top left pixel of the rectangle as displayed. Negating the height or width will flip or mirror the sprite as described in `setCrop`.

Note that this formula may produce unintuitive results if the modelview matrix **M** incorporates, for example, non-uniform scaling and/or skewing components. However, the size computation is well-defined and predictable.

Also note that in the equations above, we transform and project the X and Y axes of the sprite node as two homogeneous points each; implementations may obtain the same result by transforming them differently.

**See Also:**
Binary format

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| **`Sprite3D`**`(boolean scaled, Image2D image, Appearance appearance)`<br>Constructs a new Sprite3D with the given scaling mode, image and appearance. |

## Method Summary

| | |
|---|---|
| `Appearance` | `getAppearance()`<br>Gets the current Appearance of this Sprite3D. |
| `int` | `getCropHeight()`<br>Gets the current cropping rectangle height within the source image. |
| `int` | `getCropWidth()`<br>Gets the current cropping rectangle width within the source image. |
| `int` | `getCropX()`<br>Retrieves the current cropping rectangle X offset relative to the source image top left corner. |
| `int` | `getCropY()`<br>Retrieves the current cropping rectangle Y offset relative to the source image top left corner. |
| `Image2D` | `getImage()`<br>Gets the current Sprite3D image. |
| `boolean` | `isScaled()`<br>Returns the automatic scaling status of this Sprite3D. |
| `void` | `setAppearance(Appearance appearance)`<br>Sets the Appearance of this Sprite3D. |
| `void` | `setCrop(int cropX, int cropY, int width, int height)`<br>Sets a cropping rectangle within the source image. |
| `void` | `setImage(Image2D image)`<br>Sets the sprite image to display. |

**Methods inherited from class javax.microedition.m3g.Node**

`align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope`

**Methods inherited from class javax.microedition.m3g.Transformable**

`getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate`

**Methods inherited from class javax.microedition.m3g.Object3D**

`addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject`

## Constructor Detail

### Sprite3D

```
public Sprite3D(boolean scaled,
                Image2D image,
                Appearance appearance)
```

Constructs a new Sprite3D with the given scaling mode, image and appearance. The sprite image and appearance can be changed at any time, but the scaling mode is fixed at construction. If the appearance is null, rendering and picking of the sprite is disabled.

The crop rectangle is set such that its top left corner is at the top left corner of the image, and its width and height are equal to the dimensions of the image. However, if the width (or height) of the crop rectangle would exceed the implementation defined maximum, the width (or height) is set to the maximum value instead. The maximum crop rectangle size can be queried with `getProperties`.

**Parameters:**
`scaled` - *true* to make this Sprite3D scaled; *false* to disable scaling
`image` - pixel data and image properties to use to draw this Sprite3D
`appearance` - the Appearance to use for this Sprite3D, or null to disable this sprite initially
**Throws:**
`java.lang.NullPointerException` - if `image` is null

## Method Detail

### isScaled

```
public boolean isScaled()
```

Returns the automatic scaling status of this Sprite3D. Note that the scaling mode cannot be changed after construction.

**Returns:**
*true* if this sprite is scaled; *false* if it is unscaled

### setAppearance

```
public void setAppearance(Appearance appearance)
```

Sets the Appearance of this Sprite3D. Note that the PolygonMode, Texture and Material components of the Appearance are ignored.

**Parameters:**
`appearance` - the Appearance to set, or null to disable this sprite
**See Also:**
`getAppearance`

### getAppearance

```
public Appearance getAppearance()
```

Gets the current Appearance of this Sprite3D.

**Returns:**
the current Appearance of this sprite
**See Also:**
`setAppearance`

### setImage

```
public void setImage(Image2D image)
```

Sets the sprite image to display. The crop rectangle is reset in the same way as in the constructor.

**Parameters:**
`image` - pixel data and image properties to use to draw this sprite
**Throws:**
`java.lang.NullPointerException` - if `image` is null
**See Also:**
`getImage`

### getImage

```
public Image2D getImage()
```

Gets the current Sprite3D image.

**Returns:**
the Image2D object used to draw this sprite
**See Also:**
`setImage`

### setCrop

```
public void setCrop(int cropX,
                    int cropY,
                    int width,
                    int height)
```

Sets a cropping rectangle within the source image. This allows a subsection of the image to be used as the source for the pixels of the sprite. This can be used for selection of individual frames of animation, scrolling of captions, or other effects.

The position of the upper left corner of the crop rectangle is given in pixels, relative to the upper left corner of the Image2D. Note that the relative position may be negative in either or both axes.

If the crop rectangle has zero width or height, the sprite is not rendered or picked. If, on the other hand, the width and/or height are negative, the sprite image is flipped in the X and/or Y axes, respectively. Note that the crop rectangle remains in the same position within the source image regardless of the signs of the width and height; only the drawing order of the pixels is changed.

The absolute values of the crop width and height are limited to an implementation defined maximum that can be queried from Graphics3D.

Wrapping of the source image is not supported. If the crop rectangle lies completely or partially outside of the image boundaries, the (imaginary) pixels outside of the image are treated as if failing the alpha test. In other words, they are not rendered, but the on-screen size and pixel zoom factor of the sprite remain the same as if the crop rectangle were completely inside the image.

**Parameters:**
`cropX` - the X position of the top left of the crop rectangle, in pixels
`cropY` - the Y position of the top left of the crop rectangle, in pixels
`width` - the width of the crop rectangle, in pixels
`height` - the height of the crop rectangle, in pixels

**Throws:**
`java.lang.IllegalArgumentException` - if `width` or `height` exceeds the implementation defined maximum

## getCropX

`public int **getCropX**()`

Retrieves the current cropping rectangle X offset relative to the source image top left corner.

**Returns:**
the X offset of the cropping rectangle

**See Also:**
`setCrop`

## getCropY

`public int **getCropY**()`

Retrieves the current cropping rectangle Y offset relative to the source image top left corner.

**Returns:**
the Y offset of the cropping rectangle

**See Also:**
`setCrop`

## getCropWidth

`public int **getCropWidth**()`

Gets the current cropping rectangle width within the source image. The width may be negative, in which case the image data is flipped in the X axis.

**Returns:**

the width of the cropping rectangle

**See Also:**

`setCrop`

## getCropHeight

```
public int getCropHeight()
```

Gets the current cropping rectangle height within the source image. The height may be negative, in which case the image data is flipped in the Y axis.

**Returns:**

the height of the cropping rectangle

**See Also:**

`setCrop`

**javax.microedition.m3g**

# Class Texture2D

```
java.lang.Object
  └─ javax.microedition.m3g.Object3D
       └─ javax.microedition.m3g.Transformable
            └─ javax.microedition.m3g.Texture2D
```

public class **Texture2D**
extends Transformable

An Appearance component encapsulating a two-dimensional texture image and a set of attributes specifying how the image is to be applied on submeshes. The attributes include wrapping, filtering, blending, and texture coordinate transformation.

## Texture image data

The texture image is stored as a reference to an Image2D. The image may be in any of the formats defined in Image2D. The width and height of the image must be non-negative powers of two, but they need not be equal. The maximum allowed size for a texture image is specific to each implementation, and it can be queried with `Graphics3D.getProperties()`.

Mipmap level images are generated automatically by repeated filtering of the base level image. No particular method of filtering is mandated, but a 2x2 box filter is recommended. It is not possible for the application to supply the mipmap level images explicitly.

If the referenced Image2D is modified by the application, or a new Image2D is bound as the texture image, the modifications are immediately reflected in the Texture2D. Be aware, however, that switching to another texture image or updating the pre-existing image may trigger expensive operations, such as mipmap level image generation or (re) allocation of memory. It is therefore recommended that texture images not be updated unnecessarily.

## Texture mapping

### Transformation

The first step in applying a texture image onto a submesh is to apply the *texture transformation* to the texture coordinates of each vertex of that submesh. The transformation is defined in the Texture2D object itself, while the texture coordinates are obtained from the VertexBuffer object associated with that submesh.

The incoming texture coordinates may have either two or three components (see VertexBuffer), but for the purposes of multiplication with a 4x4 matrix they are augmented to have four components. If the third component is not given, it is implicitly set to zero. The fourth component is always assumed to be 1.

The texture transformation is very similar to the node transformation. They both consist of translation, orientation and scale components, as well as a generic 4x4 matrix component. The order of concatenating the components is the same. The only difference is that the bottom row of the matrix part must be (0 0 0 1) in case of a node transformation but not in case of a texture transformation. The methods to manipulate the individual transformation components of both node and texture transformations are defined in the base class, Transformable.

Formally, a homogeneous vector **p** = (s, t, r, 1), representing a point in texture space, is transformed to a point **p**' = (s', t', r', q') as follows:

**p**' = **TRSMp**

where **T**, **R** and **S** denote the translation, orientation and scale components, respectively, and **M** is the generic 4x4 matrix.

The translation, orientation and scale components of the texture transformation can be animated independently from each other. The matrix component is not animatable at all; it can only be changed using the `setTransform` method.

## Projection

The texture transformation described above yields the transformed texture coordinates (s', t', r', q') for each vertex of a triangle. The final texture coordinates for each rasterized fragment, in turn, are computed in two steps: interpolation and projection.

- **Interpolation.** The per-vertex texture coordinates are interpolated across the triangle to obtain the "un-projected" texture coordinate for each fragment. If the implementation supports perspective correction and the perspective correction flag in PolygonMode is enabled, this interpolation must perform some degree of perspective correction; otherwise, simple linear interpolation may (but does not have to) be used.
- **Projection.** The first three components of the interpolated texture coordinate are divided by the fourth component. Formally, the interpolated texture coordinate **p**' = (s', t', r', q') is transformed into **p**'' = (s'', t'', r'', 1) as follows:

  **p**'' = **p**'/q' = (s'/q', t'/q', r'/q', 1)

  Again, if perspective correction is either not supported or not enabled, the implementation may do the projection on a per-vertex basis and interpolate the projected values instead of the original values. Otherwise, some degree of perspective correction must be applied. Ideally, the perspective divide would be done for each fragment separately.

The r'' component of the result may be ignored, because 3D texture images are not supported in this version of the API; only the first two components are required to index a 2D image.

## Texel fetch

The transformed, interpolated and projected s'' and t'' texture coordinates of a fragment are used to fetch texel(s) from the texture image according to the selected wrapping and filtering modes.

The coordinates s'' and t'' relate to the texture image such that (0, 0) is the upper left corner of the image and (1, 1) is the lower right corner. Thus, s'' increases from left to right and t'' increases from top to bottom. The `REPEAT` and `CLAMP` texture wrapping modes define the treatment of coordinate values that are outside of the [0, 1] range.

Note that the t'' coordinate is reversed with respect to its orientation in OpenGL; however, the texture image orientation is reversed as well. As a net result, there is no difference in actual texture coordinate values between this API and OpenGL in common texturing operations. The only difference arises when rendering to a texture image that is subsequently mapped onto an object. In that case, the t texture coordinates of the object need to be reversed (t' = 1 - t). If this is not done at the modeling stage, it can be done at run-time using the texture transformation. Of course, the whole issue of texture coordinate orientation is only relevant in cases where existing OpenGL code and meshes are ported to this API.

## Texture filtering

There are two independent components in the texture filtering mode: filtering *between* mipmap levels and filtering *within* a mipmap level. There are three choices for level filtering and two choices for image filtering, yielding the six combinations listed in the table below.

| **Level filter** | **Image filter** | **Description** | **OpenGL equivalent** |
|---|---|---|---|
| `BASE_LEVEL` | `NEAREST` | Point sampling within the base level | `NEAREST` |
| `BASE_LEVEL` | `LINEAR` | Bilinear filtering within the base level | `LINEAR` |
| `NEAREST` | `NEAREST` | Point sampling within the nearest mipmap level | `NEAREST_MIPMAP_NEAREST` |
| `NEAREST` | `LINEAR` | Bilinear filtering within the nearest mipmap level | `LINEAR_MIPMAP_NEAREST` |
| `LINEAR` | `NEAREST` | Point sampling within two nearest mipmap levels | `NEAREST_MIPMAP_LINEAR` |
| `LINEAR` | `LINEAR` | Bilinear filtering within two nearest mipmap levels (trilinear filtering) | `LINEAR_MIPMAP_LINEAR` |

Only the first combination (point sampling within the base level) must be supported by all implementations. Any of the other five options may be silently ignored.

## Texture blending

The texture blending mode specifies how to combine the filtered texture color with the incoming fragment color in a texturing unit. This is equivalent to the texture environment mode in OpenGL.

The incoming fragment color $C_f = (R_f, G_f, B_f)$ and alpha $A_f$ for each texture unit are the results of texture application in the previous texture unit, or for texture unit 0, the interpolated vertex color. Similarly, the texture values $C_t$ and $A_t$ are the results of texture sampling and filtering as specified above. For luminance textures, the filtered luminance value $L_t$ is converted to an RGB color as $C_t = (L_t, L_t, L_t)$. In the `BLEND` mode, the texture blend color $C_c$, set by `setBlendColor`, is also used.

The input values are combined to output values $C_v$ and $A_v$ depending on the source texture format and the current texture blend mode as shown in the table below.

| **Texture format** | **Blending mode** | | | | |
|---|---|---|---|---|---|
| | `REPLACE` | `MODULATE` | `DECAL` | `BLEND` | `ADD` |
| `ALPHA` | $C_v = C_f$<br>$A_v = A_t$ | $C_v = C_f$<br>$A_v = A_f A_t$ | undefined | $C_v = C_f$<br>$A_v = A_f A_t$ | $C_v = C_f$<br>$A_v = A_f A_t$ |
| `LUMINANCE` | $C_v = C_t$<br>$A_v = A_f$ | $C_v = C_f C_t$<br>$A_v = A_f$ | undefined | $C_v = C_f (1 - C_t) + C_c C_t$<br>$A_v = A_f$ | $C_v = C_f + C_t$<br>$A_v = A_f$ |
| `LUM_ALPHA` | $C_v = C_t$<br>$A_v = A_t$ | $C_v = C_f C_t$<br>$A_v = A_f A_t$ | undefined | $C_v = C_f (1 - C_t) + C_c C_t$<br>$A_v = A_f A_t$ | $C_v = C_f + C_t$<br>$A_v = A_f A_t$ |

| | | | | | |
|---|---|---|---|---|---|
| RGB | $C_v = C_t$<br>$A_v = A_f$ | $C_v = C_f\,C_t$<br>$A_v = A_f$ | $C_v = C_t$<br>$A_v = A_f$ | $C_v = C_f\,(1 - C_t) + C_c\,C_t$<br>$A_v = A_f$ | $C_v = C_f + C_t$<br>$A_v = A_f$ |
| RGBA | $C_v = C_t$<br>$A_v = A_t$ | $C_v = C_f\,C_t$<br>$A_v = A_f\,A_t$ | $C_v = C_f\,(1 - A_t) + C_t\,A_t$<br>$A_v = A_f$ | $C_v = C_f\,(1 - C_t) + C_c\,C_t$<br>$A_v = A_f\,A_t$ | $C_v = C_f + C_t$<br>$A_v = A_f\,A_t$ |

## Implementation guidelines

Texturing is done according to the OpenGL 1.3 specification, section 3.8, with the following exceptions:

- 1D, 3D, and cube texture maps are not supported;
- texture borders are not supported (border width is always zero);
- The T texture coordinate and the texture image are both reversed;
- minification and magnification filters cannot be specified separately;
- mipmap level images are generated automatically by the implementation;
- LOD parameters for mipmap level selection are implementation defined;
- The `COMBINE` texture environment mode is not supported;
- The `INTENSITY` texture image format is not supported;
- The secondary color is not supported.

Texture filtering modes, other than point sampling of the base level image, are rendering quality hints that may be ignored by the implementation. However, if they are implemented, the implementation must be according to the OpenGL 1.3 specification.

**See Also:**
Binary format

| **Field Summary** | |
|---|---|
| `static int` | **`FILTER_BASE_LEVEL`**<br>A level filtering parameter to `setFiltering` that selects the base level image, even if mipmap levels exist. |
| `static int` | **`FILTER_LINEAR`**<br>A parameter to `setFiltering` that selects linear filtering. |
| `static int` | **`FILTER_NEAREST`**<br>A parameter to `setFiltering` that selects nearest neighbor filtering. |
| `static int` | **`FUNC_ADD`**<br>A parameter to `setBlending`, specifying that the texel color is to be added to the fragment color. |
| `static int` | **`FUNC_BLEND`**<br>A parameter to `setBlending`, specifying that the texture blend color is to be blended into the fragment color in proportion to the texel RGB values. |
| `static int` | **`FUNC_DECAL`**<br>A parameter to `setBlending`, specifying that the texel color is to be blended into the fragment color in proportion to the texel alpha. |

| | |
|---|---|
| static int | **FUNC_MODULATE**<br>A parameter to `setBlending`, specifying that the texel color and/or alpha are to be multiplied with the fragment color and alpha. |
| static int | **FUNC_REPLACE**<br>A parameter to `setBlending`, specifying that the texel color and/or alpha are to replace the fragment color and alpha. |
| static int | **WRAP_CLAMP**<br>A parameter to `setWrapping`, specifying that the texture image is to be repeated only once. |
| static int | **WRAP_REPEAT**<br>A parameter to `setWrapping`, specifying that the texture image is to be repeated indefinitely. |

| **Constructor Summary** |
|---|
| `Texture2D(Image2D image)`<br>Constructs a new texture object with the given image, setting the texture attributes to their default values. |

| **Method Summary** | |
|---|---|
| int | `getBlendColor()`<br>Returns the current texture blend color for this Texture2D. |
| int | `getBlending()`<br>Returns the current texture blend mode for this Texture2D. |
| Image2D | `getImage()`<br>Retrieves the current base level (full size) texture image. |
| int | `getImageFilter()`<br>Returns the current texture image filter. |
| int | `getLevelFilter()`<br>Returns the current texture level filter. |
| int | `getWrappingS()`<br>Returns the current texture wrapping mode for the S texture coordinate. |
| int | `getWrappingT()`<br>Returns the current texture wrapping mode for the T texture coordinate. |
| void | `setBlendColor(int RGB)`<br>Sets the texture blend color for this Texture2D. |
| void | `setBlending(int func)`<br>Selects the texture blend mode, or blend function, for this Texture2D. |
| void | `setFiltering(int levelFilter, int imageFilter)`<br>Selects the filtering mode for this Texture2D. |
| void | `setImage(Image2D image)`<br>Sets the given Image2D as the texture image of this Texture2D. |
| void | `setWrapping(int wrapS, int wrapT)`<br>Sets the wrapping mode for the S and T texture coordinates. |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform`, `getOrientation`, `getScale`, `getTransform`, `getTranslation`, `postRotate`, `preRotate`, `scale`, `setOrientation`, `setScale`, `setTransform`, `setTranslation`, `translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack`, `animate`, `duplicate`, `find`, `getAnimationTrack`, `getAnimationTrackCount`, `getReferences`, `getUserID`, `getUserObject`, `removeAnimationTrack`, `setUserID`, `setUserObject` |

## Field Detail

### FILTER_BASE_LEVEL

`public static final int` **`FILTER_BASE_LEVEL`**

A level filtering parameter to `setFiltering` that selects the base level image, even if mipmap levels exist. This is not applicable as the `imageFilter` parameter.

**See Also:**
Constant Field Values

### FILTER_LINEAR

`public static final int` **`FILTER_LINEAR`**

A parameter to `setFiltering` that selects linear filtering. As a level filter parameter, it specifies that a weighted average of the two closest mipmap levels should be selected. As an image filter parameter, it specifies that a weighted average of the four texels that are nearest to (s, t) in Manhattan distance should be selected.

**See Also:**
Constant Field Values

### FILTER_NEAREST

`public static final int` **`FILTER_NEAREST`**

A parameter to `setFiltering` that selects nearest neighbor filtering. As a level filter parameter, it specifies that the closest mipmap level should be selected. As an image filter parameter, it specifies that the texel that is nearest to (s, t) in Manhattan distance should be selected.

**See Also:**
Constant Field Values

### FUNC_ADD

`public static final int` **`FUNC_ADD`**

A parameter to `setBlending`, specifying that the texel color is to be added to the fragment color. The texel alpha is to be multiplied with the fragment alpha.

**See Also:**
Constant Field Values

### FUNC_BLEND

`public static final int` **`FUNC_BLEND`**

A parameter to `setBlending`, specifying that the texture blend color is to be blended into the fragment color in proportion to the texel RGB values. The texel alpha is to be multiplied with the fragment alpha.

**See Also:**
Constant Field Values

### FUNC_DECAL

`public static final int` **`FUNC_DECAL`**

A parameter to `setBlending`, specifying that the texel color is to be blended into the fragment color in proportion to the texel alpha.

**See Also:**
Constant Field Values

### FUNC_MODULATE

`public static final int` **`FUNC_MODULATE`**

A parameter to `setBlending`, specifying that the texel color and/or alpha are to be multiplied with the fragment color and alpha.

**See Also:**
Constant Field Values

### FUNC_REPLACE

`public static final int` **`FUNC_REPLACE`**

A parameter to `setBlending`, specifying that the texel color and/or alpha are to replace the fragment color and alpha.

**See Also:**
Constant Field Values

### WRAP_CLAMP

`public static final int` **`WRAP_CLAMP`**

A parameter to `setWrapping`, specifying that the texture image is to be repeated only once. This can be specified independently for the S and T texture coordinates. Formally, clamping means that the texture coordinate value is clamped to the range [0, 1]. This is equivalent to the `CLAMP` mode, with a border width of zero, in OpenGL.

**See Also:**
Constant Field Values

### WRAP_REPEAT

`public static final int` **`WRAP_REPEAT`**

A parameter to `setWrapping`, specifying that the texture image is to be repeated indefinitely. This can be specified independently for the S and T texture coordinates. Formally, repeating the image means that the integer part of the texture coordinate is ignored and only the fractional part is used. This is equivalent to the `REPEAT` mode in OpenGL.

**See Also:**
Constant Field Values

## Constructor Detail

### Texture2D

`public` **`Texture2D`**`(Image2D image)`

Constructs a new texture object with the given image, setting the texture attributes to their default values. The default values for the wrapping, filtering and blending attributes are as follows:

- wrapping S : `WRAP_REPEAT`
- wrapping T : `WRAP_REPEAT`
- level filter : `FILTER_BASE_LEVEL`
- image filter : `FILTER_NEAREST`
- blending mode : `FUNC_MODULATE`
- blend color : 0x00000000 (transparent black)

**Parameters:**
`image` - an Image2D object to set as the base level texture image

**Throws:**
`java.lang.NullPointerException` - if `image` is null
`java.lang.IllegalArgumentException` - if the width or height of `image` is not a positive power of two (1, 2, 4, 8, 16, etc.)

`java.lang.IllegalArgumentException` - if the width or height of `image` exceeds the implementation defined maximum

## Method Detail

### setImage

```
public void setImage(Image2D image)
```

Sets the given Image2D as the texture image of this Texture2D. Mipmap level images are generated automatically from the given Image2D, if and when necessary.

**Parameters:**
`image` - an Image2D object to set as the base level texture image

**Throws:**
`java.lang.NullPointerException` - if `image` is null
`java.lang.IllegalArgumentException` - if the width or height of `image` is not a positive power of two (1, 2, 4, 8, 16, etc.)
`java.lang.IllegalArgumentException` - if the width or height of `image` exceeds the implementation defined maximum

**See Also:**
`getImage`

### getImage

```
public Image2D getImage()
```

Retrieves the current base level (full size) texture image.

**Returns:**
the current base level texture image

**See Also:**
`setImage`

### setFiltering

```
public void setFiltering(int levelFilter,
                         int imageFilter)
```

Selects the filtering mode for this Texture2D. The available filtering modes are defined in the class description. Note that this setting is only a hint -- implementations may ignore it and choose a filtering method at their own discretion.

**Parameters:**
`levelFilter` - filtering between mipmap levels
`imageFilter` - filtering within a mipmap level

**Throws:**
`java.lang.IllegalArgumentException` - if `levelFilter` is not one of `FILTER_BASE_LEVEL, FILTER_NEAREST, FILTER_LINEAR`

`java.lang.IllegalArgumentException` - if `imageFilter` is not one of `FILTER_NEAREST, FILTER_LINEAR`

**See Also:**

`getLevelFilter`, `getImageFilter`

## getLevelFilter

```
public int getLevelFilter()
```

Returns the current texture level filter. Note that the set value is returned, even if the implementation only supports a subset of the available filtering methods.

**Returns:**

the current filtering between mipmap levels

**Since:**

M3G 1.1

**See Also:**

`setFiltering`

## getImageFilter

```
public int getImageFilter()
```

Returns the current texture image filter. Note that the set value is returned, even if the implementation only supports a subset of the available filtering methods.

**Returns:**

the current filtering within a mipmap level

**Since:**

M3G 1.1

**See Also:**

`setFiltering`

## setWrapping

```
public void setWrapping(int wrapS,
                        int wrapT)
```

Sets the wrapping mode for the S and T texture coordinates.

**Parameters:**

`wrapS` - S texture coordinate wrapping mode
`wrapT` - T texture coordinate wrapping mode

**Throws:**

`java.lang.IllegalArgumentException` - if `wrapS` or `wrapT` is not one of `WRAP_CLAMP, WRAP_REPEAT`

**See Also:**

`getWrappingS`, `getWrappingT`

## getWrappingS

`public int getWrappingS()`

Returns the current texture wrapping mode for the S texture coordinate.

**Returns:**
the current S coordinate wrapping mode

**See Also:**
`setWrapping`

## getWrappingT

`public int getWrappingT()`

Returns the current texture wrapping mode for the T texture coordinate.

**Returns:**
the current T coordinate wrapping mode

**See Also:**
`setWrapping`

## setBlending

`public void setBlending(int func)`

Selects the texture blend mode, or blend function, for this Texture2D. The available blending modes are defined in the class description.

**Parameters:**
`func` - the texture blending function to select

**Throws:**
`java.lang.IllegalArgumentException` - if `func` is not one of `FUNC_REPLACE, FUNC_MODULATE, FUNC_DECAL, FUNC_BLEND, FUNC_ADD`

**See Also:**
`getBlending`

## getBlending

`public int getBlending()`

Returns the current texture blend mode for this Texture2D.

**Returns:**
the current texture blending function

**See Also:**
`setBlending`

## setBlendColor

```
public void setBlendColor(int RGB)
```

Sets the texture blend color for this Texture2D. The high order byte of the color value (that is, the alpha component) is ignored.

**Parameters:**
`RGB` - the new texture blend color in 0x00RRGGBB format

**See Also:**
`getBlendColor`

## getBlendColor

```
public int getBlendColor()
```

Returns the current texture blend color for this Texture2D. The high order byte of the color value (that is, the alpha component) is guaranteed to be zero.

**Returns:**
the current texture blend color in 0x00RRGGBB format

**See Also:**
`setBlendColor`

**javax.microedition.m3g**

# Class Transform

```
java.lang.Object
  └─javax.microedition.m3g.Transform
```

public class **Transform**
extends java.lang.Object

A generic 4x4 floating point matrix, representing a transformation. By default, all methods dealing with Transform objects operate on arbitrary 4x4 matrices. Any exceptions to this rule are documented explicitly at the method level.

Even though arbitrary 4x4 matrices are generally allowed, using non-invertible (singular) matrices may produce undefined results or an arithmetic exception in some situations. Specifically, if the modelview matrix of an object is non-invertible, the results of normal vector transformation and fogging are undefined for that object.

| Constructor Summary |
|---|
| `Transform()`<br>Constructs a new Transform object and initializes it to the 4x4 identity matrix. |
| `Transform(Transform transform)`<br>Constructs a new Transform object and initializes it by copying in the contents of the given Transform. |

| Method Summary | |
|---|---|
| `void` | `get(float[] matrix)`<br>Retrieves the contents of this transformation as a 16-element float array. |
| `void` | `invert()`<br>Inverts this matrix, if possible. |
| `void` | `postMultiply(Transform transform)`<br>Multiplies this transformation from the right by the given transformation. |
| `void` | `postRotate(float angle, float ax, float ay, float az)`<br>Multiplies this transformation from the right by the given rotation matrix, specified in axis-angle form. |
| `void` | `postRotateQuat(float qx, float qy, float qz, float qw)`<br>Multiplies this transformation from the right by the given rotation matrix, specified in quaternion form. |
| `void` | `postScale(float sx, float sy, float sz)`<br>Multiplies this transformation from the right by the given scale matrix. |
| `void` | `postTranslate(float tx, float ty, float tz)`<br>Multiplies this transformation from the right by the given translation matrix. |
| `void` | `set(float[] matrix)`<br>Sets this transformation by copying from the given 16-element float array. |
| `void` | `set(Transform transform)`<br>Sets this transformation by copying the contents of the given Transform. |

| | |
|---|---|
| void | `setIdentity()`<br>Replaces this transformation with the 4x4 identity matrix. |
| void | `transform(float[] vectors)`<br>Multiplies the given array of 4D vectors with this matrix. |
| void | `transform(VertexArray in, float[] out, boolean W)`<br>Multiplies the elements of the given VertexArray with this matrix, storing the transformed values in a float array. |
| void | `transpose()`<br>Transposes this matrix. |

## Constructor Detail

### Transform

`public Transform()`

Constructs a new Transform object and initializes it to the 4x4 identity matrix.

### Transform

`public Transform(Transform transform)`

Constructs a new Transform object and initializes it by copying in the contents of the given Transform.

**Parameters:**
`transform` - the Transform object to copy the contents of
**Throws:**
`java.lang.NullPointerException` - if `transform` is null

## Method Detail

### setIdentity

`public void setIdentity()`

Replaces this transformation with the 4x4 identity matrix.

### set

`public void set(Transform transform)`

Sets this transformation by copying the contents of the given Transform. The pre-existing contents of this transformation are discarded.

**Parameters:**

`transform` - the new transformation

**Throws:**

`java.lang.NullPointerException` - if `transform` is null

### set

```
public void set(float[] matrix)
```

Sets this transformation by copying from the given 16-element float array. The pre-existing contents of this transformation are discarded. The elements in the source array are organized in *row-major* order:

```
 0   1   2   3
 4   5   6   7
 8   9  10  11
12  13  14  15
```

In other words, the second element of the source array is copied to the second element of the first row in the matrix, and so on.

**Parameters:**

`matrix` - the new transformation matrix as a flat float array

**Throws:**

`java.lang.NullPointerException` - if `matrix` is null

`java.lang.IllegalArgumentException` - if `matrix.length < 16`

### get

```
public void get(float[] matrix)
```

Retrieves the contents of this transformation as a 16-element float array. The matrix elements are copied to the array in row-major order, that is, in the same order as in the `set(float[])` method.

**Parameters:**

`matrix` - a flat float array to populate with the matrix contents

**Throws:**

`java.lang.NullPointerException` - if `matrix` is null

`java.lang.IllegalArgumentException` - if `matrix.length < 16`

### invert

```
public void invert()
```

Inverts this matrix, if possible. The contents of this transformation are replaced with the result.

**Throws:**

`java.lang.ArithmeticException` - if this transformation is not invertible

### transpose

```
public void transpose()
```

Transposes this matrix. The contents of this transformation are replaced with the result.

## postMultiply

```
public void postMultiply(Transform transform)
```

Multiplies this transformation from the right by the given transformation. The contents of this transformation are replaced with the result. Denoting this transformation by **M** and the given transformation by **T**, the new value for this transformation is computed as follows:

**M**' = **MT**

**Parameters:**
`transform` - the right-hand-side matrix multiplicant

**Throws:**
`java.lang.NullPointerException` - if `transform` is null

## postScale

```
public void postScale(float sx,
                      float sy,
                      float sz)
```

Multiplies this transformation from the right by the given scale matrix. The contents of this transformation are replaced with the result. Denoting this transformation by **M** and the scale matrix by **S**, the new value for this transformation is computed as follows:

**M**' = **MS**

The scaling factors may be non-uniform, and negative scale factors (mirroring transforms) are also allowed. The scale matrix **S** is constructed from the given scale factors (sx sy sz) follows:

```
sx  0   0   0
0   sy  0   0
0   0   sz  0
0   0   0   1
```

**Parameters:**
`sx` - scaling factor along the X axis
`sy` - scaling factor along the Y axis
`sz` - scaling factor along the Z axis

## postRotate

```
public void postRotate(float angle,
                       float ax,
                       float ay,
                       float az)
```

Multiplies this transformation from the right by the given rotation matrix, specified in axis-angle form. The contents of this transformation are replaced with the result. Denoting this transformation by **M** and the rotation matrix by **R**, the new value for this transformation is computed as follows:

**M'** = **MR**

The rotation is specified such that looking along the positive rotation axis, the rotation is `angle` degrees clockwise (or, equivalently, looking on the opposite direction of the rotation axis, the rotation is `angle` degrees counterclockwise).

The rotation matrix **R** is constructed from the given angle and axis (x y z) as follows:

```
xx(1-c)+c    xy(1-c)-zs   xz(1-c)+ys   0
yx(1-c)+zs   yy(1-c)+c    yz(1-c)-xs   0
xz(1-c)-ys   yz(1-c)+xs   zz(1-c)+c    0
     0            0            0       1
```

where c = cos(angle) and s = sin(angle). If the axis (x y z) is not unit-length, it will be normalized automatically before constructing the matrix.

**Parameters:**
`angle` - angle of rotation about the axis, in degrees
`ax` - X component of the rotation axis
`ay` - Y component of the rotation axis
`az` - Z component of the rotation axis

**Throws:**
`java.lang.IllegalArgumentException` - if the rotation axis `(ax ay az)` is zero and `angle` is nonzero

## postRotateQuat

```
public void postRotateQuat(float qx,
                           float qy,
                           float qz,
                           float qw)
```

Multiplies this transformation from the right by the given rotation matrix, specified in quaternion form. The contents of this transformation are replaced with the result. Denoting this transformation by **M** and the rotation matrix by **R**, the new value for this transformation is computed as follows:

**M'** = **MR**

The rotation matrix **R** is constructed from the given quaternion (x y z w) as follows:

```
1-(2yy+2zz)   2xy-2zw      2xz+2yw      0
  2xy+2zw   1-(2xx+2zz)    2yz-2xw      0
  2xz-2yw     2yz+2xw    1-(2xx+2yy)    0
     0           0            0         1
```

The input quaternion is normalized to a 4-dimensional unit vector prior to constructing the rotation matrix. A quaternion with a vector part of all zeros is therefore normalized to (0 0 0 1), which represents a rotation by 2*pi, that is, no rotation at all. The only illegal input condition occurs when all components of the quaternion are zero.

**Parameters:**

`qx` - X component of the quaternion's vector part
`qy` - Y component of the quaternion's vector part
`qz` - Z component of the quaternion's vector part
`qw` - scalar component of the quaternion

**Throws:**

`java.lang.IllegalArgumentException` - if all quaternion components are zero

## postTranslate

```
public void postTranslate(float tx,
                          float ty,
                          float tz)
```

Multiplies this transformation from the right by the given translation matrix. The contents of this transformation are replaced with the result. Denoting this transformation by **M** and the translation matrix by **T**, the new value for this transformation is computed as follows:

**M**' = **MT**

The translation matrix **T** is constructed from the given translation vector (tx ty tz) as follows:

```
1   0   0   tx
0   1   0   ty
0   0   1   tz
0   0   0   1
```

**Parameters:**

`tx` - X component of the translation vector
`ty` - Y component of the translation vector
`tz` - Z component of the translation vector

## transform

```
public void transform(VertexArray in,
                      float[] out,
                      boolean W)
```

Multiplies the elements of the given VertexArray with this matrix, storing the transformed values in a float array.

The input VertexArray may have any number of elements (E), two or three components per element (C), and any component size (8 or 16 bit). The float array is filled in with E elements, each having four components. The multiplication is always done with a full 4x4 matrix and all four components of the result are returned.

The implied value of the missing fourth component (W) of each input element depends on the boolean parameter. If the parameter is set to *true*, the W components of all vectors are set to 1.0 prior to multiplication. If the parameter is *false*, the W components are set to 0.0. If the elements of the input array have only two components, the missing third component is always set to zero.

This method does not take into account the scale and bias that may be associated with vertex positions and texture coordinates. (This is simply because the scale and bias values are defined in VertexBuffer, not VertexArray.) If the application wishes to use this method for transforming the vertex positions in a specific VertexBuffer, for example, the scale and bias can be applied to this Transform directly. See the code fragment below for how to do that.

Note that this is a simple matrix-by-vector multiplication; no division by W or other operations are implied. The interpretation of the input and output values is up to each application.

**Parameters:**

`in` - a VertexArray of 2D or 3D vectors to multiply with this matrix
`out` - a 4D float array to populate with the transformed vectors
`W` - *true* to set the W component of each input vector implicitly to 1.0; *false* to set them to 0.0

**Throws:**

`java.lang.NullPointerException` - if `in` is null
`java.lang.NullPointerException` - if `out` is null
`java.lang.IllegalArgumentException` - if `in.numComponents == 4`
`java.lang.IllegalArgumentException` - if `out.length < 4*E`, where E is the number of elements in the input VertexArray

**Example:**

**A method for transforming a vertex coordinate array.**

```
void transformPoints(Transform t, VertexBuffer vb, float[] out)
 {
     // Make a copy of the given Transform so that we can restore
     // its original contents at the end.

     Transform tmp = new Transform(t);

     // Retrieve the vertex coordinate array and its associated
     // scale and bias. In real applications, the float array
     // and the temporary Transform object should both be class
     // variables to avoid garbage collection.

     float[] scaleBias = new float[4];
     VertexArray points = vb.getPositions(scaleBias);

     // Note the order of constructing the transformation matrix.
     // The coordinates must be scaled first, then biased:
     //   v' = T S v

     t.postTranslate(scaleBias[1], scaleBias[2], scaleBias[3]);
     t.postScale(scaleBias[0], scaleBias[0], scaleBias[0]);
     t.transform(points, out, true);

     // Restore the original Transform.

     t.set(tmp);
 }
```

## transform

`public void` **`transform`**`(float[] vectors)`

Multiplies the given array of 4D vectors with this matrix. The transformation is done in place, that is, the original vectors are overwritten with the transformed vectors.

The vectors are given as a flat array of (x y z w) quadruplets. The length of the array divided by 4 gives the number of vectors to transform.

Note that this is a simple matrix-by-vector multiplication; no division by W or other operations are implied. The interpretation of the input and output values is up to each application.

**Parameters:**

`vectors` - the vectors to transform, in (xyzw xyzw xyzw ...) order

**Throws:**

`java.lang.NullPointerException` - if `vectors` is null

`java.lang.IllegalArgumentException` - if `vectors.length % 4 != 0`

**javax.microedition.m3g**

# Class Transformable

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.Transformable
```

**Direct Known Subclasses:**

Node, Texture2D

public abstract class **Transformable**
extends Object3D

An abstract base class for Node and Texture2D, defining common methods for manipulating node and texture transformations.

Node transformations and texture transformations consist of four individual components: translation (**T**), orientation (**R**), scale (**S**) and a generic 4x4 matrix (**M**). Formally, a homogeneous vector **p** = (x, y, z, w), representing vertex coordinates (in Node) or texture coordinates (in Texture2D), is transformed into **p**' = (x', y', z', w') as follows:

**p**' = **TRSMp**

See the Node and Texture2D class descriptions for more information on node transformations and texture transformations.

## Instantiation

Transformable is an abstract class, and therefore has no public constructor. When a class derived from Transformable is instantiated, the attributes inherited from it will have the following default values:

- scale : (1,1,1)
- translation : (0,0,0)
- orientation : angle = 0, axis = undefined
- matrix : identity

The transformation components are initially set to identity so that they do not affect the texture coordinates or vertex coordinates in any way. Note that the orientation axis can be left undefined because the angle is zero.

**See Also:**

Binary format

| **Method Summary** | |
|---|---|
| void | `getCompositeTransform(Transform transform)` Retrieves the composite transformation matrix of this Transformable. |

| | |
|---|---|
| `void` | **`getOrientation`**`(float[] angleAxis)`<br>Retrieves the orientation component of this Transformable. |
| `void` | **`getScale`**`(float[] xyz)`<br>Retrieves the scale component of this Transformable. |
| `void` | **`getTransform`**`(Transform transform)`<br>Retrieves the matrix component of this Transformable. |
| `void` | **`getTranslation`**`(float[] xyz)`<br>Retrieves the translation component of this Transformable. |
| `void` | **`postRotate`**`(float angle, float ax, float ay, float az)`<br>Multiplies the current orientation component from the right by the given orientation. |
| `void` | **`preRotate`**`(float angle, float ax, float ay, float az)`<br>Multiplies the current orientation component from the left by the given orientation. |
| `void` | **`scale`**`(float sx, float sy, float sz)`<br>Multiplies the current scale component by the given scale factors. |
| `void` | **`setOrientation`**`(float angle, float ax, float ay, float az)`<br>Sets the orientation component of this Transformable. |
| `void` | **`setScale`**`(float sx, float sy, float sz)`<br>Sets the scale component of this Transformable. |
| `void` | **`setTransform`**`(Transform transform)`<br>Sets the matrix component of this Transformable by copying in the given Transform. |
| `void` | **`setTranslation`**`(float tx, float ty, float tz)`<br>Sets the translation component of this Transformable. |
| `void` | **`translate`**`(float tx, float ty, float tz)`<br>Adds the given offset to the current translation component. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Method Detail

### setOrientation

```
public void setOrientation(float angle,
                           float ax,
                           float ay,
                           float az)
```

Sets the orientation component of this Transformable. The orientation is specified such that looking along the rotation axis, the rotation is `angle` degrees clockwise. Note that the axis does not have to be a unit vector.

**Parameters:**
`angle` - angle of rotation about the axis, in degrees
`ax` - X component of the rotation axis
`ay` - Y component of the rotation axis
`az` - Z component of the rotation axis

**Throws:**
`java.lang.IllegalArgumentException` - if the rotation axis `(ax ay az)` is zero and `angle` is nonzero

**See Also:**
`getOrientation`, `preRotate`, `postRotate`

### preRotate

```
public void preRotate(float angle,
                      float ax,
                      float ay,
                      float az)
```

Multiplies the current orientation component from the left by the given orientation. The orientation is given in axis-angle format, as in `setOrientation`.

Denoting the given orientation by **R**' and the current orientation by **R**, the new orientation is computed as follows:

**R**'' = **R**' **R**

Depending on the internal representation of orientations, the multiplication may be done with quaternions, matrices, or something else, as long as the resulting orientation is the same as it would be if matrices or quaternions were used.

**Parameters:**
`angle` - angle of rotation about the axis, in degrees
`ax` - X component of the rotation axis
`ay` - Y component of the rotation axis
`az` - Z component of the rotation axis

**Throws:**
`java.lang.IllegalArgumentException` - if the rotation axis `(ax ay az)` is zero and `angle` is nonzero

**See Also:**
`setOrientation`, `postRotate`

### postRotate

```
public void postRotate(float angle,
                       float ax,
                       float ay,
                       float az)
```

Multiplies the current orientation component from the right by the given orientation. Except for the multiplication order, this method is equivalent to `preRotate`.

Denoting the given orientation by **R**' and the current orientation by **R**, the new orientation is computed as follows:

**R**'' = **RR**'

**Parameters:**

`angle` - angle of rotation about the axis, in degrees
`ax` - X component of the rotation axis
`ay` - Y component of the rotation axis
`az` - Z component of the rotation axis

**Throws:**

`java.lang.IllegalArgumentException` - if the rotation axis `(ax ay az)` is zero and `angle` is nonzero

**See Also:**

`setOrientation`, `preRotate`

## getOrientation

```
public void getOrientation(float[] angleAxis)
```

Retrieves the orientation component of this Transformable.

The returned axis and angle values are not necessarily the same that were last written to the orientation component. Instead, they may be any values that produce an equivalent result. For example, a 90 degree rotation about the positive Z axis is equivalent to a 270 degree rotation about the negative Z axis. In particular, if the rotation angle is zero, the returned rotation axis is undefined and may also be the zero vector.

**Parameters:**

`angleAxis` - a float array to fill in with `(angle ax ay az)`

**Throws:**

`java.lang.NullPointerException` - if `angleAxis` is null
`java.lang.IllegalArgumentException` - if `angleAxis.length < 4`

**See Also:**

`setOrientation`

## setScale

```
public void setScale(float sx,
                     float sy,
                     float sz)
```

Sets the scale component of this Transformable.

Note that if any of the scale factors are set to zero, this transformation becomes uninvertible. That, in turn, may cause certain operations to produce undefined results or to fail with an ArithmeticException.

**Parameters:**

`sx` - scale along the X axis
`sy` - scale along the Y axis
`sz` - scale along the Z axis

**See Also:**

`getScale`, `scale`

## scale

```
public void scale(float sx,
                  float sy,
                  float sz)
```

Multiplies the current scale component by the given scale factors. Denoting the current scale by (sx sy sz) and the given scale by (sx' sy' sz'), the new scale factors are computed as follows:

sx'' = sx * sx'
sy'' = sy * sy'
sz'' = sz * sz'

Since this is an operation on scalar values, the order of multiplication makes no difference. Unlike with the rotation component, separate methods for left and right multiplication are therefore not needed.

**Parameters:**
`sx` - scale along the X axis
`sy` - scale along the Y axis
`sz` - scale along the Z axis
**See Also:**
`setScale`

## getScale

```
public void getScale(float[] xyz)
```

Retrieves the scale component of this Transformable.

**Parameters:**
`xyz` - a float array to fill in with `(sx sy sz)`
**Throws:**
`java.lang.NullPointerException` - if `xyz` is null
`java.lang.IllegalArgumentException` - if `xyz.length < 3`
**See Also:**
`setScale`

## setTranslation

```
public void setTranslation(float tx,
                           float ty,
                           float tz)
```

Sets the translation component of this Transformable.

**Parameters:**
`tx` - translation along the X axis
`ty` - translation along the Y axis

`tz` - translation along the Z axis

**See Also:**

`getTranslation`, `translate`

## translate

```
public void translate(float tx,
                      float ty,
                      float tz)
```

Adds the given offset to the current translation component. Denoting the current translation component by (tx ty tz) and the given offset by (tx' ty' tz'), the new translation component is computed as follows:

tx" = tx + tx'
ty" = ty + ty'
tz" = tz + tz'

**Parameters:**

`tx` - translation along the X axis
`ty` - translation along the Y axis
`tz` - translation along the Z axis

**See Also:**

`setTranslation`

## getTranslation

```
public void getTranslation(float[] xyz)
```

Retrieves the translation component of this Transformable.

**Parameters:**

`xyz` - a float array to fill in with `(tx ty tz)`

**Throws:**

`java.lang.NullPointerException` - if `xyz` is null
`java.lang.IllegalArgumentException` - if `xyz.length < 3`

**See Also:**

`setTranslation`

## setTransform

```
public void setTransform(Transform transform)
```

Sets the matrix component of this Transformable by copying in the given Transform. This does not affect the separate translation, orientation and scale components.

A generic matrix component is required for transformations that can not be expressed in the component form efficiently, or at all. These include, for example, pivot transforms and non-axis-aligned scales.

If this Transformable is a Node object, the bottom row of the given matrix must be (0 0 0 1). Projective transformations are not supported in the scene graph so as to reduce run-time memory consumption and to accelerate rendering. Note, however, that arbitrary 4x4 modelview matrices are supported in the immediate mode.

**Parameters:**

`transform` - the Transform object to copy in, or null to indicate the identity matrix

**Throws:**

`java.lang.IllegalArgumentException` - if this Transformable is a Node and the bottom row of `transform` is not (0 0 0 1)

**See Also:**

`getTransform`

## getTransform

`public void getTransform(Transform transform)`

Retrieves the matrix component of this Transformable. This does not include the separate translation, orientation and scale components. The transformation is copied into the given Transform object.

**Parameters:**

`transform` - the Transform object to receive the transformation matrix

**Throws:**

`java.lang.NullPointerException` - if `transform` is null

**See Also:**

`setTransform`

## getCompositeTransform

`public void getCompositeTransform(Transform transform)`

Retrieves the composite transformation matrix of this Transformable. The composite transformation matrix is the concatenation of the translation, rotation, scale and generic matrix components. Formally, **C** = **T R S M**. The composite transformation is copied into the given Transform object.

**Parameters:**

`transform` - the Transform object to receive the composite transformation matrix

**Throws:**

`java.lang.NullPointerException` - if `transform` is null

**javax.microedition.m3g**

## Class TriangleStripArray

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.IndexBuffer
          └ javax.microedition.m3g.TriangleStripArray
```

public class **TriangleStripArray**
extends IndexBuffer

TriangleStripArray defines an array of *triangle strips*. In a triangle strip, the first three vertex indices define the first triangle. Each subsequent index together with the two previous indices defines a new triangle. For odd triangles, two of the indices must be swapped to produce correct winding. The first triangle is considered even. For example, the strip S = (2, 0, 1, 4) defines two triangles: (2, 0, 1) and (1, 0, 4).

Triangle strip indices may be explicitly defined, as in the example above, or they may be implicit. In an implicit TriangleStripArray, only the first index of the first strip is specified. All subsequent indices are one greater than their predecessor. For example, if there are two strips with lengths 3 and 4, and the initial index is 10, the strips are formed as follows: $S_1$ = (10, 11, 12) and $S_2$ = (13, 14, 15, 16).

Triangle strips may contain so-called degenerate triangles, that is, triangles that have zero area. These are completely valid input to the API. The implementation must take the necessary steps to ensure that degenerate triangles do not produce any rasterizable fragments.

Degenerate triangles often occur in explicit triangle strips that are constructed by merging multiple strips into one at the content authoring stage. Merging of strips requires that the same index be repeated two or three times in a row.

Degenerate triangles may also result from multiple vertices in the associated VertexBuffer having the same coordinates (in screen space, or already before that).

**See Also:**
Binary format

| Constructor Summary |
|---|
| `TriangleStripArray(int[] indices, int[] stripLengths)` Constructs a triangle strip array with *explicit* indices. |
| `TriangleStripArray(int firstIndex, int[] stripLengths)` Constructs a triangle strip array with *implicit* indices. |

| Methods inherited from class javax.microedition.m3g.IndexBuffer |
|---|
| `getIndexCount, getIndices` |

| Methods inherited from class javax.microedition.m3g.Object3D |
|---|

`addAnimationTrack`, `animate`, `duplicate`, `find`, `getAnimationTrack`, `getAnimationTrackCount`, `getReferences`, `getUserID`, `getUserObject`, `removeAnimationTrack`, `setUserID`, `setUserObject`

## Constructor Detail

### TriangleStripArray

```
public TriangleStripArray(int firstIndex,
                          int[] stripLengths)
```

Constructs a triangle strip array with *implicit* indices. The first index of the first strip is specified, along with the lengths of the individual strips.

**Parameters:**

`firstIndex` - index of the initial vertex of the first strip
`stripLengths` - array of per-strip vertex counts to be copied in

**Throws:**

`java.lang.NullPointerException` - if `stripLengths` is null
`java.lang.IllegalArgumentException` - if `stripLengths` is empty
`java.lang.IllegalArgumentException` - if any element in `stripLengths` is less than 3
`java.lang.IndexOutOfBoundsException` - if `firstIndex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstIndex + sum(stripLengths) > 65535`

### TriangleStripArray

```
public TriangleStripArray(int[] indices,
                          int[] stripLengths)
```

Constructs a triangle strip array with *explicit* indices. An array of indices is given, along with the lengths of the individual strips. The combined length of the strips must not exceed the number of elements in the index array. The contents of both arrays are copied in.

**Parameters:**

`indices` - array of indices to be copied in
`stripLengths` - array of per-strip index counts to be copied in

**Throws:**

`java.lang.NullPointerException` - if `indices` is null
`java.lang.NullPointerException` - if `stripLengths` is null
`java.lang.IllegalArgumentException` - if `stripLengths` is empty
`java.lang.IllegalArgumentException` - if any element in `stripLengths` is less than 3
`java.lang.IllegalArgumentException` - if `indices.length < sum(stripLengths)`
`java.lang.IndexOutOfBoundsException` - if any element in `indices` is negative
`java.lang.IndexOutOfBoundsException` - if any element in `indices` is greater than 65535

**javax.microedition.m3g**

## Class VertexArray

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.VertexArray
```

public class **VertexArray**
extends Object3D

An array of integer vectors representing vertex positions, normals, colors, or texture coordinates.

VertexArray objects are referenced by VertexBuffer objects. Each VertexArray may be referenced by any number of VertexBuffers, or even multiple times by the same VertexBuffer. The role in which the array is referenced determines the interpretation of the vertex attributes contained in it. For example, if a VertexArray is referenced as a normal vector array, the array entries are interpreted as 3D vectors. If the same array is referenced as a texture coordinate array, the entries are interpreted as 3D points.

Certain restrictions exist on the types of data that can be used in each role. The restrictions are as follows:

- Vertex positions must have 3 components.
- Normal vectors must have 3 components.
- Texture coordinates must have 2 or 3 components.
- Colors must have 3 or 4 components, one byte each.

**See Also:**
Binary format

| Constructor Summary |
|---|
| `VertexArray(int numVertices, int numComponents, int componentSize)`<br>Constructs a new VertexArray with the given dimensions. |

| Method Summary | |
|---|---|
| `void` | `get(int firstVertex, int numVertices, byte[] values)`<br>Returns a range of 8-bit vertex attributes. |
| `void` | `get(int firstVertex, int numVertices, short[] values)`<br>Returns a range of 16-bit vertex attributes. |
| `int` | `getComponentCount()`<br>Returns the number of components per vertex. |
| `int` | `getComponentType()`<br>Returns the data type (size) of vertex components. |
| `int` | `getVertexCount()`<br>Returns the number of vertices in this array. |

| | |
|---|---|
| void | `set(int firstVertex, int numVertices, byte[] values)`<br>Copies in an array of 8-bit vertex attributes. |
| void | `set(int firstVertex, int numVertices, short[] values)`<br>Copies in an array of 16-bit vertex attributes. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### VertexArray

```
public VertexArray(int numVertices,
                   int numComponents,
                   int componentSize)
```

Constructs a new VertexArray with the given dimensions. The array elements are initialized to zero. The elements can be set later with either the 8-bit or the 16-bit version of the `set` method, depending on the component size selected here.

**Parameters:**

`numVertices` - number of vertices in this VertexArray; must be [1, 65535]
`numComponents` - number of components per vertex; must be [2, 4]
`componentSize` - number of bytes per component; must be [1, 2]

**Throws:**

`java.lang.IllegalArgumentException` - if any of the parameters are outside of their allowed ranges

## Method Detail

### set

```
public void set(int firstVertex,
                int numVertices,
                short[] values)
```

Copies in an array of 16-bit vertex attributes. Positions, normals, and texture coordinates can be set with this method, but colors must be set with 8-bit input. This method is available only if `componentSize`, specified in the constructor, is 2.

The vertex attributes are copied in starting from the first element of the source array. The number of elements copied in is `numComponents * numVertices`, where `numComponents` is either 2, 3, or 4, as specified at construction time. The source array must have at least that many elements.

**Parameters:**

`firstVertex` - index of the first vertex to replace
`numVertices` - number of vertices to replace
`values` - array of 16-bit integers to copy vertex attributes from

**Throws:**

`java.lang.NullPointerException` - if `values` is null
`java.lang.IllegalStateException` - if this is not a 16-bit VertexArray
`java.lang.IllegalArgumentException` - if `numVertices < 0`
`java.lang.IllegalArgumentException` - if `values.length < numVertices * getComponentCount`
`java.lang.IndexOutOfBoundsException` - if `firstVertex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex + numVertices > getVertexCount`

## set

```
public void set(int firstVertex,
                int numVertices,
                byte[] values)
```

Copies in an array of 8-bit vertex attributes. All vertex attributes can be set with this method, including positions, normals, colors, and texture coordinates. This method is available only if `componentSize`, specified in the constructor, is 1.

The vertex attributes are copied in as specified in the other `set` variant.

**Parameters:**

`firstVertex` - index of the first vertex to replace
`numVertices` - number of vertices to replace
`values` - array of 8-bit integers to copy vertex attributes from

**Throws:**

`java.lang.NullPointerException` - if `values` is null
`java.lang.IllegalStateException` - if this is not an 8-bit VertexArray
`java.lang.IllegalArgumentException` - if `numVertices < 0`
`java.lang.IllegalArgumentException` - if `values.length < numVertices * getComponentCount`
`java.lang.IndexOutOfBoundsException` - if `firstVertex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex + numVertices > getVertexCount`

## getVertexCount

```
public int getVertexCount()
```

Returns the number of vertices in this array.

**Returns:**

the number of vertices

**Since:**

M3G 1.1

## getComponentCount

```
public int getComponentCount()
```

Returns the number of components per vertex.

**Returns:**
the number of components
**Since:**
M3G 1.1

## getComponentType

```
public int getComponentType()
```

Returns the data type (size) of vertex components.

**Returns:**
the number of bytes per component; 1 for bytes, 2 for shorts
**Since:**
M3G 1.1

## get

```
public void get(int firstVertex,
                int numVertices,
                short[] values)
```

Returns a range of 16-bit vertex attributes. The values are returned in the same format as in the respective `set` method.

**Parameters:**
`firstVertex` - index of the first vertex to get
`numVertices` - number of vertices to get
`values` - array of 16-bit integers to copy vertex attributes to
**Throws:**
`java.lang.NullPointerException` - if `values` is null
`java.lang.IllegalStateException` - if this is not a 16-bit VertexArray
`java.lang.IllegalArgumentException` - if `numVertices < 0`
`java.lang.IllegalArgumentException` - if `values.length < numVertices * getComponentCount`
`java.lang.IndexOutOfBoundsException` - if `firstVertex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex + numVertices > getVertexCount`
**Since:**
M3G 1.1

## get

```
public void get(int firstVertex,
                int numVertices,
                byte[] values)
```

Returns a range of 8-bit vertex attributes. The values are returned in the same format as in the respective `set` method.

**Parameters:**

`firstVertex` - index of the first vertex to get
`numVertices` - number of vertices to get
`values` - array of 8-bit integers to copy vertex attributes to

**Throws:**

`java.lang.NullPointerException` - if `values` is null
`java.lang.IllegalStateException` - if this is not an 8-bit VertexArray
`java.lang.IllegalArgumentException` - if `numVertices < 0`
`java.lang.IllegalArgumentException` - if `values.length < numVertices * getComponentCount`
`java.lang.IndexOutOfBoundsException` - if `firstVertex < 0`
`java.lang.IndexOutOfBoundsException` - if `firstVertex + numVertices > getVertexCount`

**Since:**

M3G 1.1

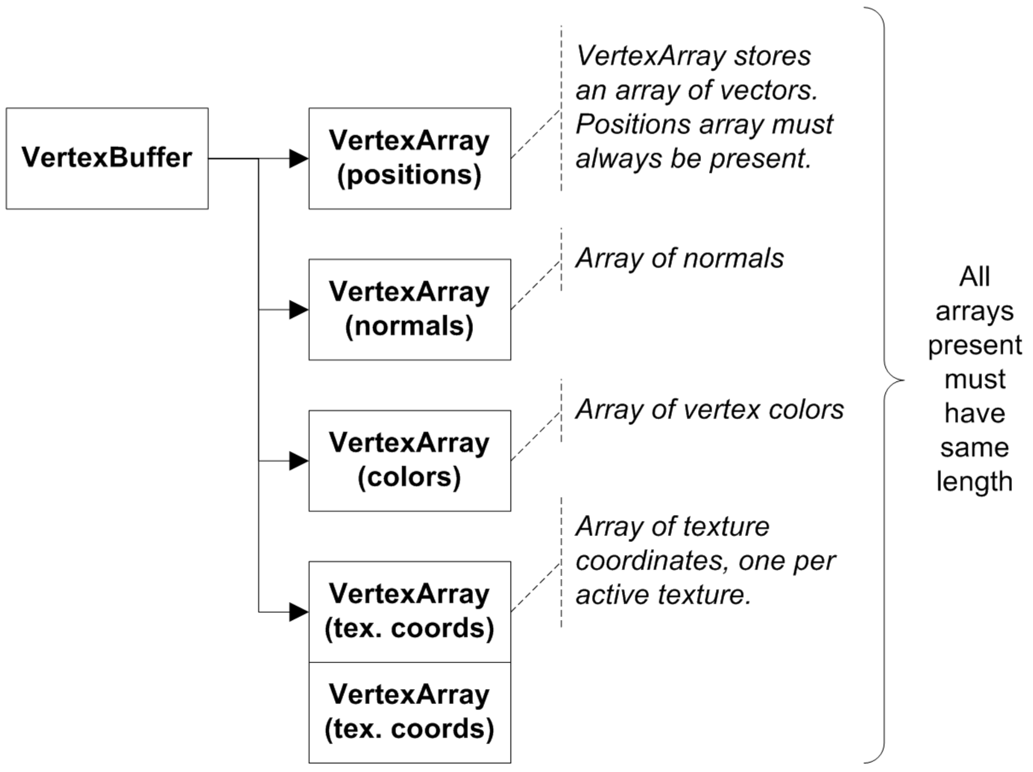
**javax.microedition.m3g**

## Class VertexBuffer

```
java.lang.Object
  └ javax.microedition.m3g.Object3D
      └ javax.microedition.m3g.VertexBuffer
```

public class **VertexBuffer**
extends Object3D

VertexBuffer holds references to VertexArrays that contain the positions, colors, normals, and texture coordinates for a set of vertices. The elements of these arrays are called *vertex attributes* in the rest of this documentation. The structure of a VertexBuffer object is shown in the figure below.


VertexBuffer
VertexArray (positions)
VertexArray (normals)
VertexArray (colors)
VertexArray (tex. coords)
VertexArray (tex. coords)
VertexArray stores an array of vectors. Positions array must always be present.
Array of normals
Array of vertex colors
Array of texture coordinates, one per active texture.
All arrays present must have same length

There can be at most one position array, one color array, and one normal array in a VertexBuffer. The number of texture coordinate arrays, however, can be anything between zero and the number of texturing units supported by the implementation, which can be queried with `getProperties`.

All vertex attribute arrays must be the same length; this is enforced by the `set` methods. The first array that is added to a previously empty VertexBuffer can have any number of elements. This is also the case if the sole previously set array is replaced with another. Any subsequently added arrays must have the same length as the first.

Vertex positions, texture coordinates, and normals are interpreted as homogeneous (4D) coordinates, where the fourth component is implicitly 1 for positions and texcoords, and 0 for normals. In other words, positions and texcoords are interpreted as 3D points, whereas normals are treated as 3D vectors. In the case of 2D texcoords, the third component is implicitly zero.

3D texture coordinates are supported, even though 3D texture maps are not. This allows some clever rendering tricks, such as cheap environment mapping by using normal vectors as texture coordinates.

### Deferred exceptions

All vertex attribute arrays are initialized to null. The application can also set them to null at any time. This is a completely legal state, as long as the VertexBuffer is not rendered or tested for intersection with a pick ray. When rendering, null vertex attributes are treated as follows:

- If the position array is null, an exception is thrown.
- If the normal array is null, and lighting is enabled, the normal vectors are undefined.
- If a texcoord array is null, and the corresponding texturing unit is enabled, the texture coordinates are undefined.
- If the color array is null, the default color is used instead.

Lighting can be disabled for a submesh by setting a null Material in Appearance. Similarly, a particular texturing unit can be turned off by setting its Texture2D to null.

When picking, null vertex attributes are treated as follows:

- If the position array is null, an exception is thrown.
- If the normal array is null, the returned normal is undefined.
- If a texcoord array is null, the returned texcoords are undefined.

The color array and the default color are ignored when picking.

**See Also:**
Binary format

| **Constructor Summary** |
|---|
| `VertexBuffer()`<br>Creates an empty VertexBuffer with default values. |

| **Method Summary** | |
|---|---|
| `VertexArray` | `getColors()`<br>Gets the current color array, or null if per-vertex colors are not set. |
| `int` | `getDefaultColor()`<br>Retrieves the default color of this VertexBuffer. |
| `VertexArray` | `getNormals()`<br>Gets the current normal vector array, or null if normals are not set. |
| `VertexArray` | `getPositions(float[] scaleBias)`<br>Returns the current vertex position array, or null if positions are not set. |
| `VertexArray` | `getTexCoords(int index, float[] scaleBias)`<br>Gets the current texture coordinate array for the specified texturing unit, or null if texture coordinates for that unit are not set. |
| `int` | `getVertexCount()`<br>Retrieves the current number of vertices in this VertexBuffer. |

| | |
|---|---|
| void | **setColors**(VertexArray colors)<br>Sets the per-vertex colors for this VertexBuffer. |
| void | **setDefaultColor**(int ARGB)<br>Sets the color to use in absence of per-vertex colors. |
| void | **setNormals**(VertexArray normals)<br>Sets the normal vectors for this VertexBuffer. |
| void | **setPositions**(VertexArray positions, float scale, float[] bias)<br>Sets the vertex positions for this VertexBuffer. |
| void | **setTexCoords**(int index, VertexArray texCoords, float scale, float[] bias)<br>Sets the texture coordinates for the specified texturing unit. |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### VertexBuffer

`public` **`VertexBuffer`**`()`

Creates an empty VertexBuffer with default values. The default values are:

- vertex count : 0
- vertex position array : null
- texture coordinate array(s) : null
- normal array : null
- color array : null (use default color)
- default color : 0xFFFFFFFF (opaque white)

## Method Detail

### getVertexCount

`public int` **`getVertexCount`**`()`

Retrieves the current number of vertices in this VertexBuffer. This is the same as the number of vertices in any of the associated VertexArrays, because they must all have the same length. If there are no VertexArrays currently in this VertexBuffer, the number of vertices is defined to be zero.

**Returns:**
the number of vertices currently in this VertexBuffer, or zero if no vertex arrays are set

## setPositions

```
public void setPositions(VertexArray positions,
                         float scale,
                         float[] bias)
```

Sets the vertex positions for this VertexBuffer. Vertex positions are specified with a 3-component VertexArray. The components are interpreted as coordinates in (X, Y, Z) order, each component being a signed 8-bit or 16-bit integer. Vertex positions have associated with them a uniform scale and a per-component bias, which are common to all vertices in the VertexArray. The final position **v'** of a vertex is computed from the original position **v** as follows:

$$\mathbf{v}' = s\mathbf{v} + \mathbf{b}$$

where *s* is the the uniform scale and **b** is the bias vector. For example, the application can map the 8-bit integers [-128, 127] to the real number range [-1, 1] by setting the scale to 2/255 and the bias to 1/255.

Non-uniform scaling is not supported due to implementation constraints. A uniform scale factor introduces no per-vertex processing overhead, as implementations may combine the scale with the transformation from object space to world space or camera space. Combining a non-uniform scale with that transformation, in contrast, would distort the normal vectors and thereby cause undesirable side effects in lighting.

**Parameters:**

`positions` - a VertexArray with 3-component vertex positions, or null to disable vertex positions
`scale` - a constant uniform scale factor common to all vertex positions
`bias` - a constant (X, Y, Z) offset to add to vertex positions after scaling, or null to set a zero bias for all components

**Throws:**

`java.lang.IllegalArgumentException` - if `(positions != null) && (positions.getComponentCount != 3)`
`java.lang.IllegalArgumentException` - if `(positions != null) && (positions.getVertexCount != getVertexCount) && (at least one other VertexArray is set)`
`java.lang.IllegalArgumentException` - if `(positions != null) && (bias != null) && (bias.length < 3)`

**See Also:**

`getPositions`

## setTexCoords

```
public void setTexCoords(int index,
                         VertexArray texCoords,
                         float scale,
                         float[] bias)
```

Sets the texture coordinates for the specified texturing unit. Texture coordinates are specified with a 2- or 3-component VertexArray. The components are interpreted in (S, T) or (S, T, R) order, each component being a signed 8-bit or 16-bit integer. Texture coordinates have associated with them a uniform scale and a per-component bias, which behave exactly the same way as with vertex positions (see `setPositions`). Non-uniform scaling is not supported, so as to make texture coordinates behave consistently with vertex positions.

**Parameters:**

`index` - index of the texturing unit to assign these texture coordinates to
`texCoords` - a VertexArray with 2- or 3-component texture coordinates, or null to disable texture coordinates for the specified unit
`scale` - a constant uniform scale factor common to all texture coordinates
`bias` - a constant (X, Y, Z) offset to add to texture coordinates after scaling, or null to set a zero bias for all components

**Throws:**

`java.lang.IllegalArgumentException` - if `(texCoords != null) && (texCoords.getComponentCount != {2,3})`
`java.lang.IllegalArgumentException` - if `(texCoords != null) && (texCoords.getVertexCount != getVertexCount) && (at least one other VertexArray is set)`
`java.lang.IllegalArgumentException` - if `(texCoords != null) && (bias != null) && (bias.length < texCoords.getComponentCount)`
`java.lang.IndexOutOfBoundsException` - if `index != [0,N]` where `N` is the implementation specific maximum texturing unit index

**See Also:**

`getTexCoords`

## setNormals

```
public void setNormals(VertexArray normals)
```

Sets the normal vectors for this VertexBuffer. The scale and bias terms are not specified for normals. Instead, the components of the normals are mapped to [-1, 1] such that the maximum positive integer maps to +1, the maximum negative integer to -1, and the mapping is linear in between. Note that the number zero, for instance, cannot be represented accurately with this scheme.

The normal vectors need not be unit length on input; they are automatically normalized prior to using them in the lighting computations. This implicit normalization does not modify the original values in the VertexArray.

**Parameters:**

`normals` - a VertexArray with 3-component normal vectors, or null to disable normals

**Throws:**

`java.lang.IllegalArgumentException` - if `(normals != null) && (normals.getComponentCount != 3)`
`java.lang.IllegalArgumentException` - if `(normals != null) && (normals.getVertexCount != getVertexCount) && (at least one other VertexArray is set)`

**See Also:**

`getNormals`

## setColors

```
public void setColors(VertexArray colors)
```

Sets the per-vertex colors for this VertexBuffer. The given VertexArray containing the color values must have either 3 (RGB) or 4 (RGBA) components per element, and the component size must be 8 bits. With RGB colors, the alpha component is implicitly set to 1 for all vertices.

The scale and bias terms are not specified for colors. Instead, color components are interpreted as unsigned integers between [0, 255], where 255 represents the maximum brightness (1.0). This is equivalent to having a scale of 1/255 and a bias of 128/255 for all components (the bias is needed because bytes in Java are always interpreted as signed values).

**Parameters:**

`colors` - a VertexArray with RGB or RGBA color values, or null to use the default color instead

**Throws:**

`java.lang.IllegalArgumentException` - if `(colors != null) && (colors.getComponentType != 1)`

`java.lang.IllegalArgumentException` - if `(colors != null) && (colors.getComponentCount != {3,4})`

`java.lang.IllegalArgumentException` - if `(colors != null) && (colors.getVertexCount != getVertexCount) && (at least one other VertexArray is set)`

**See Also:**

`getColors`

## getPositions

```
public VertexArray getPositions(float[] scaleBias)
```

Returns the current vertex position array, or null if positions are not set. The current scale and bias values are copied into the given array. If positions are not set, the scale and bias values are undefined. The first four elements of the array are overwritten with the scale and bias, in that order. Any other elements in the array are left untouched. If the given array is null, only the VertexArray is returned.

**Parameters:**

`scaleBias` - a float array to populate with the current scale (1 entry) and bias (3 entries), or null to just return the VertexArray

**Returns:**

the current VertexArray for vertex positions, or null

**Throws:**

`java.lang.IllegalArgumentException` - if `(scaleBias != null) && (scaleBias.length < 4)`

**See Also:**

`setPositions`

## getTexCoords

```
public VertexArray getTexCoords(int index,
                                float[] scaleBias)
```

Gets the current texture coordinate array for the specified texturing unit, or null if texture coordinates for that unit are not set. The current scale and bias values are copied into the given array. If the texture coordinate array is null, the scale and bias values are undefined. The first 3 or 4 elements of the array are overwritten with the scale and bias, in that order. Any other elements in the array are left untouched. The number of elements written is equal to the number of components in the returned VertexArray, plus one for the scale. If the given array is null, only the VertexArray is returned.

**Parameters:**

`index` - index of the texturing unit to get the texture coordinates of

`scaleBias` - a float array to populate with the current scale (1 entry) and bias (2 or 3 entries), or null to just return the VertexArray

**Returns:**

VertexArray with the texture coordinates for the given texturing unit, or null

**Throws:**

`java.lang.IllegalArgumentException` - if `(scaleBias != null) && (scaleBias.length < texCoords.getComponentCount+1)`

`java.lang.IndexOutOfBoundsException` - if `index != [0,N]` where `N` is the implementation specific maximum texturing unit index

**See Also:**

`setTexCoords`

## getNormals

```
public VertexArray getNormals()
```

Gets the current normal vector array, or null if normals are not set.

**Returns:**

the current VertexArray for vertex normals, or null

**See Also:**

`setNormals`

## getColors

```
public VertexArray getColors()
```

Gets the current color array, or null if per-vertex colors are not set.

**Returns:**

the current VertexArray for vertex colors, or null

**See Also:**

`setColors`

## setDefaultColor

```
public void setDefaultColor(int ARGB)
```

Sets the color to use in absence of per-vertex colors. This color will be assigned to each vertex by default. If per-vertex colors are specified, this color is ignored.

**Parameters:**

`ARGB` - the default vertex color in 0xAARRGGBB format

**See Also:**

`getDefaultColor`

## getDefaultColor

```
public int getDefaultColor()
```

Retrieves the default color of this VertexBuffer.

**Returns:**

the default vertex color in 0xAARRGGBB format

**See Also:**

`setDefaultColor`

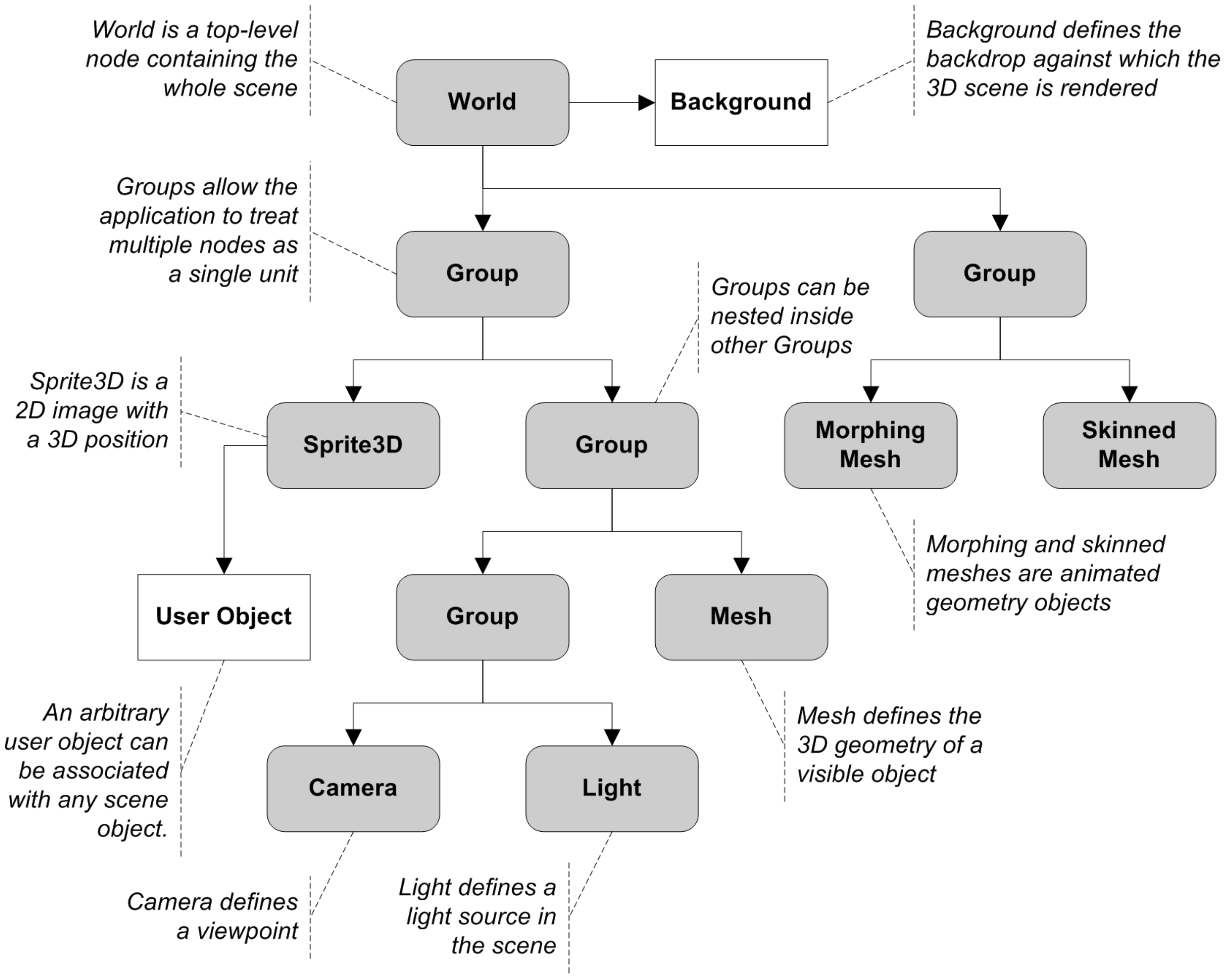
**javax.microedition.m3g**

## Class World

```
java.lang.Object
  └─ javax.microedition.m3g.Object3D
       └─ javax.microedition.m3g.Transformable
            └─ javax.microedition.m3g.Node
                 └─ javax.microedition.m3g.Group
                      └─ javax.microedition.m3g.World
```

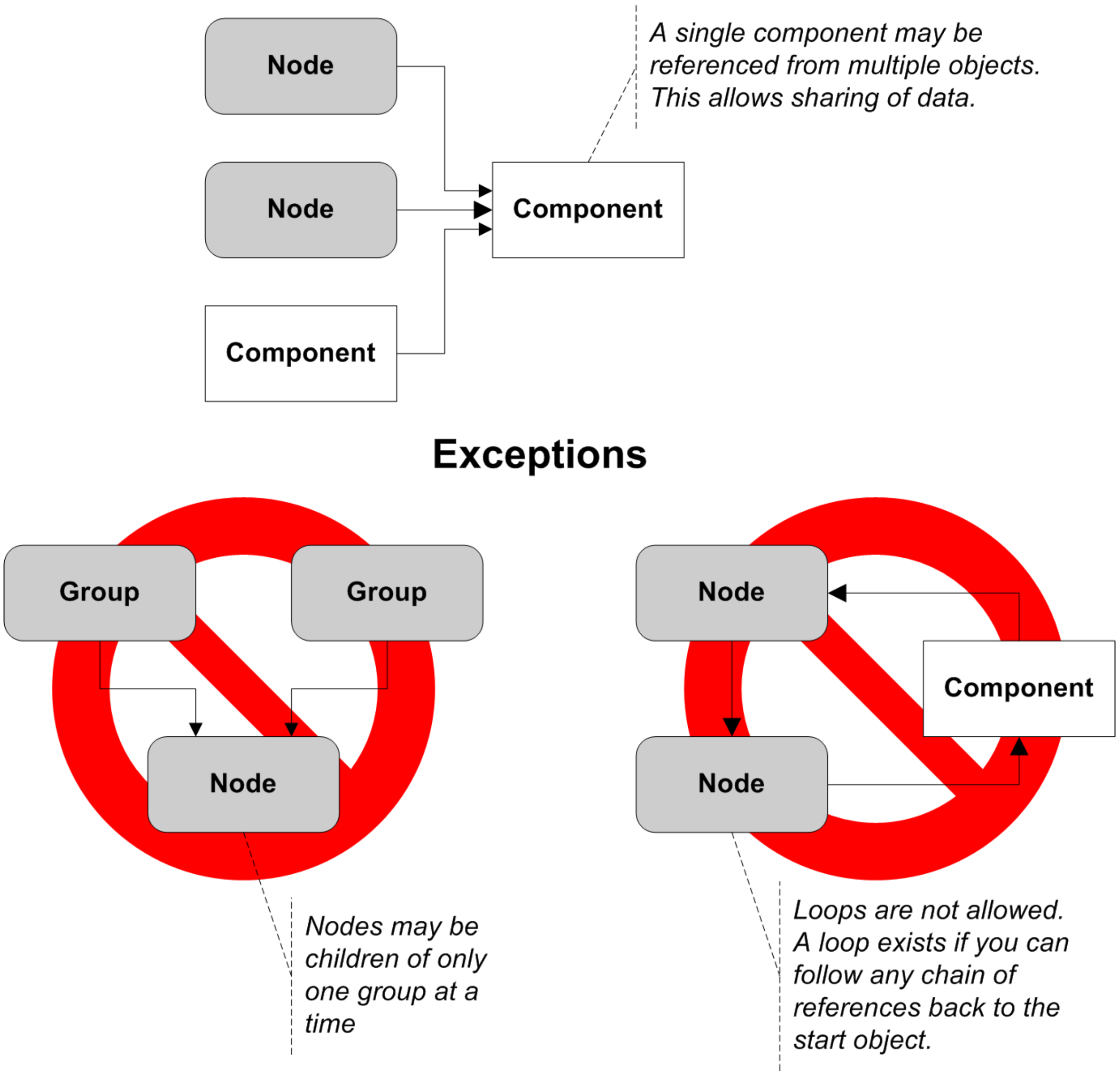
public class **World**
extends Group

A special Group node that is a top-level container for scene graphs. A scene graph is constructed from a hierarchy of nodes. In a complete scene graph, all nodes are ultimately connected to each other via a common root, which is a World node. An example of a complete scene graph is shown in the figure below.


World is a top-level node containing the whole scene
World
Background
Background defines the backdrop against which the 3D scene is rendered
Groups allow the application to treat multiple nodes as a single unit
Group
Groups can be nested inside other Groups
Group
Sprite3D is a 2D image with a 3D position
Sprite3D
Group
Morphing Mesh
Skinned Mesh
Morphing and skinned meshes are animated geometry objects
User Object
Group
Mesh
An arbitrary user object can be associated with any scene object.
Camera
Light
Mesh defines the 3D geometry of a visible object
Camera defines a viewpoint
Light defines a light source in the scene

Note that a scene graph need not be complete in order to be rendered; individual nodes and branches can be rendered using a separate method in Graphics3D. However, the semantics of rendering an incomplete scene graph are slightly different compared to rendering a World; see Graphics3D for more information.

Despite that it is called a graph, the scene graph is actually a tree structure. This implies that a node can belong to at most one group at a time, and cycles are prohibited. However, component objects, such as VertexArrays, may be referenced by an arbitrary number of nodes and components. The basic rules for building valid scene graphs are summarized below.


Node
Node
Component
Component
A single component may be referenced from multiple objects. This allows sharing of data.
Exceptions
Group
Group
Node
Nodes may be children of only one group at a time
Node
Component
Node
Loops are not allowed. A loop exists if you can follow any chain of references back to the start object.

Even though World is a scene graph node, its special role as the singular root node has two noteworthy consequences. Firstly, a World can not be a child of any Node. Secondly, the node transformation is ignored for World objects when rendering. In all other respects (get, set, animate), the transformation behaves just like any other node transformation. Note also that there is no conceptual "Universe" coordinate system above the World, contrary to some other scene graph APIs.

The method `render(World)` in Graphics3D renders a World as observed by the currently active camera of that world. If the active camera is null, or the camera is not in the world, an exception is thrown. The world can still be rendered with the `render(Node, Transform)` method by treating the World as a Group. In that case, however, the application must explicitly clear the background and set up the camera and lights prior to rendering.

**See Also:**

Binary format

## Field Summary

| **Fields inherited from class javax.microedition.m3g.Node** |
|---|
| `NONE, ORIGIN, X_AXIS, Y_AXIS, Z_AXIS` |

## Constructor Summary

| |
|---|
| **`World()`**<br>Creates an empty World with default values. |

## Method Summary

| | |
|---|---|
| `Camera` | **`getActiveCamera()`**<br>Gets the currently active camera. |
| `Background` | **`getBackground()`**<br>Retrieves the background settings of this World. |
| `void` | **`setActiveCamera`**`(Camera camera)`<br>Sets the Camera to use when rendering this World. |
| `void` | **`setBackground`**`(Background background)`<br>Sets the Background object for this World. |

| **Methods inherited from class javax.microedition.m3g.Group** |
|---|
| `addChild, getChild, getChildCount, pick, pick, removeChild` |

| **Methods inherited from class javax.microedition.m3g.Node** |
|---|
| `align, getAlignmentReference, getAlignmentTarget, getAlphaFactor, getParent, getScope, getTransformTo, isPickingEnabled, isRenderingEnabled, setAlignment, setAlphaFactor, setPickingEnable, setRenderingEnable, setScope` |

| **Methods inherited from class javax.microedition.m3g.Transformable** |
|---|
| `getCompositeTransform, getOrientation, getScale, getTransform, getTranslation, postRotate, preRotate, scale, setOrientation, setScale, setTransform, setTranslation, translate` |

| **Methods inherited from class javax.microedition.m3g.Object3D** |
|---|
| `addAnimationTrack, animate, duplicate, find, getAnimationTrack, getAnimationTrackCount, getReferences, getUserID, getUserObject, removeAnimationTrack, setUserID, setUserObject` |

## Constructor Detail

### World

`public World()`

Creates an empty World with default values. The default values are:

- background : null (clear to black)
- active camera : null (the world is not renderable)

## Method Detail

### setBackground

`public void setBackground(Background background)`

Sets the Background object for this World. The background is used for clearing the frame buffer prior to rendering the World when `Graphics3D.render(World)` is called.

If the background object is null, the default values are used. That is, the color buffer is cleared to transparent black, and the depth buffer to the maximum depth (1.0).

**Parameters:**
`background` - attributes for clearing the frame buffer, or null to use defaults
**See Also:**
`getBackground`

### getBackground

`public Background getBackground()`

Retrieves the background settings of this World.

**Returns:**
the current attributes for clearing the frame buffer
**See Also:**
`setBackground`

### setActiveCamera

`public void setActiveCamera(Camera camera)`

Sets the Camera to use when rendering this World. At the time of rendering, the camera must also be a descendant of this World.

**Parameters:**

`camera` - the Camera object to set as the active camera

**Throws:**

`java.lang.NullPointerException` - if `camera` is null

**See Also:**

`getActiveCamera`

## getActiveCamera

```
public Camera getActiveCamera()
```

Gets the currently active camera.

**Returns:**

the camera that is currently used to render this World

**See Also:**

`setActiveCamera`

# Constant Field Values

## Contents

## javax.microedition.*

### javax.microedition.m3g.AnimationTrack

| | |
|---|---|
| public static final int ALPHA | 256 |
| public static final int AMBIENT_COLOR | 257 |
| public static final int COLOR | 258 |
| public static final int CROP | 259 |
| public static final int DENSITY | 260 |
| public static final int DIFFUSE_COLOR | 261 |
| public static final int EMISSIVE_COLOR | 262 |
| public static final int FAR_DISTANCE | 263 |
| public static final int FIELD_OF_VIEW | 264 |
| public static final int INTENSITY | 265 |
| public static final int MORPH_WEIGHTS | 266 |
| public static final int NEAR_DISTANCE | 267 |
| public static final int ORIENTATION | 268 |
| public static final int PICKABILITY | 269 |
| public static final int SCALE | 270 |
| public static final int SHININESS | 271 |
| public static final int SPECULAR_COLOR | 272 |
| public static final int SPOT_ANGLE | 273 |
| public static final int SPOT_EXPONENT | 274 |
| public static final int TRANSLATION | 275 |
| public static final int VISIBILITY | 276 |

### javax.microedition.m3g.Background

| | |
|---|---|
| public static final int BORDER | 32 |
| public static final int REPEAT | 33 |

| javax.microedition.m3g.Camera | |
|---|---|
| public static final int GENERIC | 48 |
| public static final int PARALLEL | 49 |
| public static final int PERSPECTIVE | 50 |

| javax.microedition.m3g.CompositingMode | |
|---|---|
| public static final int ALPHA | 64 |
| public static final int ALPHA_ADD | 65 |
| public static final int MODULATE | 66 |
| public static final int MODULATE_X2 | 67 |
| public static final int REPLACE | 68 |

| javax.microedition.m3g.Fog | |
|---|---|
| public static final int EXPONENTIAL | 80 |
| public static final int LINEAR | 81 |

| javax.microedition.m3g.Graphics3D | |
|---|---|
| public static final int ANTIALIAS | 2 |
| public static final int DITHER | 4 |
| public static final int OVERWRITE | 16 |
| public static final int TRUE_COLOR | 8 |

| javax.microedition.m3g.Image2D | |
|---|---|
| public static final int ALPHA | 96 |
| public static final int LUMINANCE | 97 |
| public static final int LUMINANCE_ALPHA | 98 |
| public static final int RGB | 99 |
| public static final int RGBA | 100 |

| javax.microedition.m3g.KeyframeSequence | |
|---|---|
| public static final int CONSTANT | 192 |
| public static final int LINEAR | 176 |
| public static final int LOOP | 193 |
| public static final int SLERP | 177 |
| public static final int SPLINE | 178 |
| public static final int SQUAD | 179 |

| | |
|---|---|
| public static final int STEP | 180 |

| javax.microedition.m3g.Light | |
|---|---|
| public static final int AMBIENT | 128 |
| public static final int DIRECTIONAL | 129 |
| public static final int OMNI | 130 |
| public static final int SPOT | 131 |

| javax.microedition.m3g.Material | |
|---|---|
| public static final int AMBIENT | 1024 |
| public static final int DIFFUSE | 2048 |
| public static final int EMISSIVE | 4096 |
| public static final int SPECULAR | 8192 |

| javax.microedition.m3g.Node | |
|---|---|
| public static final int NONE | 144 |
| public static final int ORIGIN | 145 |
| public static final int X_AXIS | 146 |
| public static final int Y_AXIS | 147 |
| public static final int Z_AXIS | 148 |

| javax.microedition.m3g.PolygonMode | |
|---|---|
| public static final int CULL_BACK | 160 |
| public static final int CULL_FRONT | 161 |
| public static final int CULL_NONE | 162 |
| public static final int SHADE_FLAT | 164 |
| public static final int SHADE_SMOOTH | 165 |
| public static final int WINDING_CCW | 168 |
| public static final int WINDING_CW | 169 |

| javax.microedition.m3g.Texture2D | |
|---|---|
| public static final int FILTER_BASE_LEVEL | 208 |
| public static final int FILTER_LINEAR | 209 |
| public static final int FILTER_NEAREST | 210 |
| public static final int FUNC_ADD | 224 |
| public static final int FUNC_BLEND | 225 |

| | |
|---|---|
| `public static final int FUNC_DECAL` | 226 |
| `public static final int FUNC_MODULATE` | 227 |
| `public static final int FUNC_REPLACE` | 228 |
| `public static final int WRAP_CLAMP` | 240 |
| `public static final int WRAP_REPEAT` | 241 |

# File Format for Mobile 3D Graphics API

## Abstract

This specification defines a 3D Graphics File Format that complements the Mobile 3D Graphics API (M3G). The file format is provided as a compact and standardised way of populating a scene graph.

## Contents

---

## 1 Important Notes

The data here are *not* serialized by Java's own serialization mechanism. They are serialized by the M3G serialization mechanism, which produces and loads data streams conforming to the M3G file format specification.

For more details of the mechanisms for loading a M3G compliant file, please refer to the documentation for the Loader class.

## 2 MIME Type and File Extension

The MIME type for this file format is `application/m3g`. The file extension (for systems that do not support MIME type queries) is `.m3g`, to match the lowest level name in the package hierarchy.

## 3 Data Types

### 3.1 Fundamental Data Types

There are several data types which are regarded as fundamental. These are as follows:

| Type Name | Description |
|---|---|
| `Byte` | A single, unsigned 8-bit byte. |
| `Int16` | A signed 16 bit value, stored as two bytes, lower byte first. |
| `UInt16` | An unsigned 16 bit value, stored as two bytes, lower byte first. |
| `Int32` | A signed 32 bit value, stored as four bytes, lowest byte first. |
| `UInt32` | An unsigned 32 bit value, stored as four bytes, lowest byte first. |
| `Float32` | A single precision floating point value, in 32-bit format as defined by IEEE-754. This is stored as four bytes, with the least significant byte of the mantissa first, and the exponent byte last.<br>Note that only normal numeric values and positive 0 can be stored. Special values such as denormals, infinities, NaNs, negative 0, and indefinite values are disallowed and must be treated as errors. |
| `String` | A null-terminated Unicode string, coded as UTF-8. |
| `Boolean` | A single byte with the value 0 (false) or 1 (true). Other values are disallowed and must be treated as errors. |

## 3.2 Compound Data Types

In order to avoid having to repeatedly specify sequences of the same types many times, some compound data types are defined for convenience. The composition of these is listed to show both their makeup and the order in which the simple elements are to be serialized. These are as follows:

| Type Name | Description | Composition |
|---|---|---|
| Vector3D | A 3D vector. | `Float32 x;`<br>`Float32 y;`<br>`Float32 z;` |
| Matrix | A 4x4 generalized matrix. The 16 elements of the matrix are output in the same order as they are retrieved using the API Transform.get method. In other words, in this order:<br>`0  1  2  3`<br>`4  5  6  7`<br>`8  9  10 11`<br>`12 13 14 15` | `Float32 elements [16];` |
| ColorRGB | A color, with no alpha information. Each component is scaled so that 0x00 is 0.0, and 0xFF is 1.0. | `Byte red;`<br>`Byte green;`<br>`Byte blue;` |

| | | |
|---|---|---|
| ColorRGBA | A color, with alpha information. Each component is scaled so that 0x00 is 0.0, and 0xFF is 1.0. The alpha value is scaled so that 0x00 is completely transparent, and 0xFF is completely opaque. | `Byte red;` `Byte green;` `Byte blue;` `Byte alpha;` |
| ObjectIndex | The index of a previously encountered object in the file. Although this is serialized as a single unsigned integer, it is included in the compound type list because of the additional semantic information embodied in its type. A value of 0 is reserved to indicate a null reference; actual object indices start from 1. Object indices must refer only to null or to an object which has already been created during the input deserialization of a file - they must be less than or equal to the index of the object in which they appear. Other values are disallowed and must be treated as errors. | `UInt32 index;` |
| *Type*[] | A variable-length array of any type is always output in a counted form, with the count first. Each element is then output in index order, starting from 0. The last element has index (count-1). If the array is empty, then only a 0 count is output. | `UInt32 count;` *Type* `arrayValue [0];` *Type* `arrayValue [1];` *...etc.* |
| *Type*[*count*] | Arrays with an explicit length are either always have the same constant number of elements, or this count is specified elsewhere, so only the elements are output. Each element is then output in index order, starting from 0. The last element has index (length-1). If the array is empty, then nothing is output. | *Type* `arrayValue [0];` *Type* `arrayValue [1];` *...etc.* |

## 4 File Structure

The file consists of the file identifier, followed by one or more sections. Thus the overall file structure looks like this:

| | |
|---|---|
| | File Identifier |
| Section 0 | File Header Object |
| Section 1 | External Reference Objects |
| Section 2 | Scene Objects |
| Section 3 | Scene Objects |
| ... | ... |
| Section *n* | Scene Objects |

The reason for having different sections is that some of the objects, such as the mesh objects, should be compressed to reduce file size, whereas other objects, such as the header object, should not be compressed. The header object must be kept uncompressed since it should be easy to read quickly.

The first section, Section 0, must be present, must be uncompressed and must contain only the header object. This object contains information about the file as a whole, and is discussed in detail in Section 10.1.

If there are external references in the file, then these must all appear in a single section immediately following the header section. This section may be compressed or uncompressed. External references allow scenes to be built up from a collection of separate files, and are discussed in detail in Section 10.2.

Following these are an unspecified number of sections containing scene objects.

The file must contain the header section, plus at least one other non-empty section (containing at least one object). It is possible to have a file consisting solely of external references, or solely of scene objects.

A file containing no objects at all is not a valid M3G file, and must be treated as an error.

## 5 File Identifier

The file identifier is a unique set of bytes that will differentiate the file from other types of files. It consists of 12 bytes, as follows:

```
Byte[12] FileIdentifier = { 0xAB, 0x4A, 0x53, 0x52, 0x31, 0x38,
0x34, 0xBB, 0x0D, 0x0A, 0x1A, 0x0A }
```

This can also be expressed using C-style character definitions as:

```
Byte[12] FileIdentifier = { '«', 'J', 'S', 'R', '1', '8', '4', '»',
'\r', '\n', '\x1A', '\n' }
```

The rationale behind the choice values in the identifier is based on the rationale for the identifier in the PNG specification. This identifier both identifies the file as a M3G file and provides for immediate detection of common file-transfer problems.

- Byte [0] is chosen as a non-ASCII value to reduce the probability that a text file may be misrecognized as a M3G file.
- Byte [0] also catches bad file transfers that clear bit 7.
- Bytes [1..6] identify the format, and are the ascii values for the string "JSR184".
- Byte [7] is for aesthetic balance with byte 1 (they are a matching pair of double-angle quotation marks).
- Bytes [8..9] form a CR-LF sequence which catches bad file transfers that alter newline sequences.
- Byte [10] is a control-Z character, which stops file display under MS-DOS, and further reduces the chance that a text file will be falsely recognised.
- Byte [11] is a final line feed, which checks for the inverse of the CR-LF translation problem.

A decoder may further verify that the next byte is 0 (this is the first byte of the mandatory uncompressed header section). This will catch bad transfers that drop or alter zero bytes.

# 6 Section

A section is a data container for one or more objects. The section header determines if the objects are compressed or not, how much object data there is, and also contains a checksum.

In this document, we will talk about "sections that are compressed" and "sections that are uncompressed". In reality, we will mean "sections where the objects are compressed", and "sections where the objects are uncompressed".

Each section has the following structure:

```
Byte                          CompressionScheme
UInt32                        TotalSectionLength
UInt32                        UncompressedLength
Byte[TotalSectionLength-13] Objects
UInt32                        Checksum
```

We will now go through the individual parts of the section.

## 6.1 CompressionScheme

This field tells how the `Objects` field in this section is compressed. It also specifies what checksum algorithm is used. Currently, only the Adler32 checksum is mandatory. Compression only applies to the `Object` data, and not to the other fields in the section.

`CompressionScheme` must be one of the following values:

| | |
|---|---|
| 0 | Uncompressed, Adler32 Checksum |
| 1 | ZLib compression, 32 k buffer size, Adler32 Checksum |
| 2...255 | Reserved |

Example:

```
Byte CompressionScheme = 1;
```

indicates that the `Objects` field in the section is compressed using zlib with 32 k buffer size.

The values 2...255 are reserved for future releases and are disallowed. A loader that follows the specification must report an error if they are found.

## 6.2 TotalSectionLength

This is the total length of the section in bytes; from the start of this section to the start of the next section.

Example:

```
UInt32 TotalSectionLength = 2056
```

indicates that this section, including the `CompressionScheme`, `TotalSectionLength`, `UncompressedLength`, `Objects` and `Checksum` fields, will be 2056 bytes in length.

## 6.3 UncompressedLength

Knowing the size of the decompressed data ahead of time can be used to make Zlib inflation much easier and less memory hungry. Therefore, the size of the compressed part of the section (in bytes) before compression (or after decompression) is serialized as part of the section information. Since it is only the `Objects` field that can be compressed, `UncompressedLength` contains the length of the `Objects` field after decompression. If no compression is specified for this section, this equals the actual number of bytes serialized in the `Objects` array.

A value of 0 in this field is legal - the section is simply ignored. However, it is recommended that any process that creates a file should check for 0 length sections and eliminate them to reduce file size.

Example:

```
UInt32 UncompressedLength = 4560
```

Means that in this section, after decompression, the `Objects` field is 4560 bytes in length.

## 6.4 Objects

The objects in each section are serialized as an array of bytes, one after the other. This array of bytes will either be compressed (if CompressionScheme is 1) or it will be uncompressed. If it is compressed, it is compressed as a single chunk of data, not as separate objects. Zero bits must be padded in the end to make the `Objects` field byte aligned.

The structure of each individual object's data is documented in Section 10 and Section 11.

## 6.5 Checksum

To be able to verify that the section was correctly loaded, there is a 32-bit checksum of all the data in the section. The checksum algorithm is specified by the `CompressionScheme` field. Currently, only the Adler32 checksum is mandatory. The checksum is calculated using all preceding bytes in the section, i.e. the `CompressionScheme`, `TotalSectionLength`, `UncompressedLength`, and the actual serialized data in the `Objects` field (i.e. in its compressed form if compression is specified).

Example:

```
UInt32 Checksum = 0xffe806a3
```

On limited devices, we might not be able to afford to load an entire section before interpreting it. Thus the loader may start interpreting the objects before knowing that the section as a whole is correct. However, the checksums are still useful in that we at least know afterwards that there was an otherwise undetected error if the checksums differed.

Even on a system that can afford to load an entire section before loading it, it is possible to have errors in the file. The content creation program can have a defect, the transmission of the file could be error-prone, or the file could have been altered as part of a deliberate attack on the device. Thus it is important that the loader tries to detect errors also in files that have correct checksums.

The loader implementation may decide not to compute (and/or check) the checksum. Thus, a file with erroneous checksums is not guaranteed to be rejected. However, a file with erroneous checksums is not a M3G compliant file and must not pass a strict verification test.

# 7 Object Structure

The object data stored in each section is first decompressed and then interpreted as a sequence of objects. This separates the act of decompression from the interpretation of the data. All data documented in this section is assumed already to be in its uncompressed form.

Each object in the file represents one object in the scene graph tree, and is stored in a chunk. The structure of an object chunk is as follows:

```
Byte          ObjectType
UInt32        Length
Byte[Length]  Data
```

## 7.1 ObjectType

This field describes what type of object has been serialized. For instance, we could have a Camera node, a Mesh node or a Texture2D object. Section 12 includes a table that shows the correspondence between ObjectType values and the actual object types. The ObjectType field must hold a valid value as defined in Section 12. The reserved object types (values 23..254) must be treated as errors.

The values 0 and 0xFF are special: 0 represents the header object, and 0xFF represents an external reference.

Example:

```
Byte ObjectType = 14
```

This means that the current object is a `Mesh` object (see Section 12).

## 7.2 Length

This contains the length of the Data array, in bytes. Note that a value of 0 in this field may be legal; some objects require no additional data over and above their mere presence.

Example:

```
UInt32 Length = 2032
```

indicates that the `Data` field of this object spans 2032 bytes in the (decompressed) file.

## 7.3 Data

This is data that is specific for the object. It is up to the loader to interpret this data according to the object type and populate the object accordingly. Detailed information on the data for each object type is documented in Section 10 and Section 11.

For instance, if the object just contained a single color, the Data would be a 3 byte long array, where the first byte represents the red component, the second byte the green component, and the third byte the blue component.

Attempts to read off the end of an object's data are disallowed and must be signalled as errors. An example of this would be an object with a reported length of 32 bytes, but which internally specifies an array with 65537 members.

Conversely, the deserialization code for each object may also check that each byte of the data belongs to a valid interpretation. Additional bytes after the end of an object's valid data are disallowed. This condition may be difficult to determine on the target platform, but any file which contains "extra" data in object chunks is not a M3G compliant file and must not pass a strict verification test.

## 8 Object Ordering

All the objects in the scene graph are serialized in leaf-first order, or reference based order as it is also called. Before serializing a specific object, all other objects referenced by that object must already have been serialized. Objects may refer to themselves if this is allowed by the scene data structures.

By definition, the root of the tree will be sent last.

Note that cycles are not allowed in the file format. There is one special case where they *are* allowed in the run-time scene graph, namely Node alignment. Before a scene graph containing cyclic references can be written into a file, the cycles must be broken. This can be done by inserting dummy target nodes as children of the original alignment targets. For example, if a leaf node is aligned to the World, an empty Group with an identity transformation is inserted as a child of the World, and the alignment redirected to that.

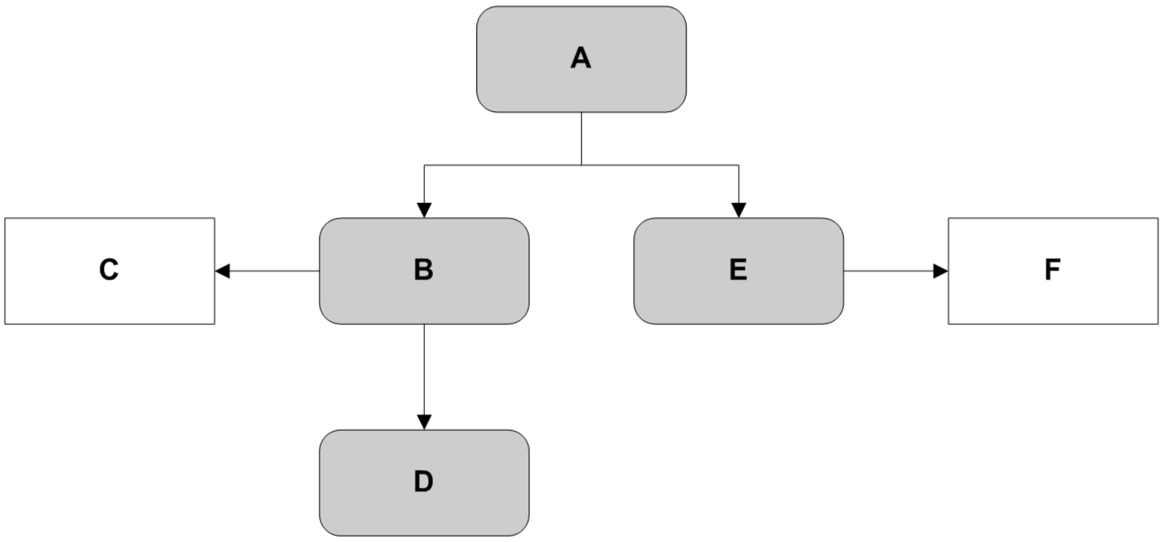
Given a scene graph with no cycles, it is possible to use a "leaves first" strategy for output - start by serializing all the objects that do not reference other objects, and then all the objects that refer to the objects already sent, and so it continues until all objects are sent.

Alternatively, a "depth first" strategy can be used, where each object recursively applies the following procedure, to build up a table of references in the correct order. (It is assumed that the table is initially empty.)

```
BuildReferenceTable:
for each reference in this object,
    call BuildReferenceTable on the referred object
if this object is not already in the reference table,
    append this object to the reference table.
```

Each object can then be serialized from the reference table in order.

For example, assume that we have the following tree structure:

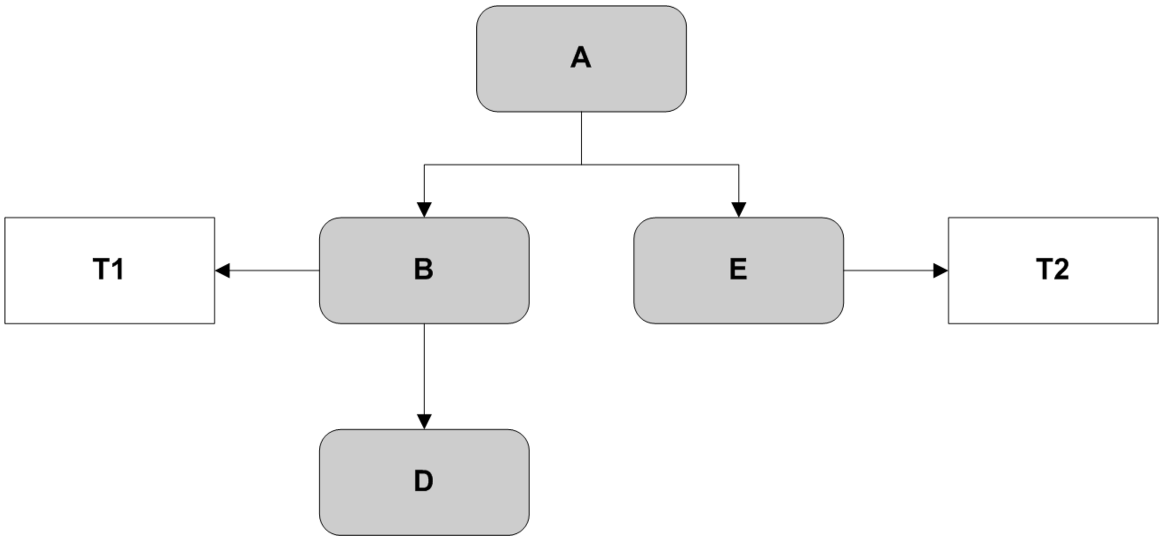


One valid ordering of the objects is C D F B E A. This is the ordering that occurs if the "leaves first" method is used. Note that other leaf-first orderings are also valid, for instance F D C E B A.

The "depth-first" method produces valid orderings where interior nodes in the graph may be sent before all the leaves have been sent. An ordering produced by the depth-first method discussed above might be C D B F E A.

The only important thing is that any objects referenced by a particular object are sent before the object itself.

With this flexibility, the ordering of references can be forced by the file creator if this is advantageous. For example, if we wish textures to be sent in a separate section that is uncompressed. Thus, if we have the following tree:



where T1 and T2 are textures, we can send the scene graph using, for instance:

| Identifier | | File Identifier (see Section 5) |
|---|---|---|
| Section 0 | Uncompressed | File Header Object |
| Section 1 | Uncompressed | T1 T2 |
| Section 2 | Compressed | D B E A |

Other orderings are also possible, for instance:

| Identifier | | File Identifier (see Section 5) |
|---|---|---|
| Section 0 | Uncompressed | File Header Object |
| Section 1 | Uncompressed | T1 T2 |
| Section 2 | Compressed | D E B A |

or even (with a naive file creator):

| Identifier | | File Identifier (see Section 5) |
|---|---|---|
| Section 0 | Uncompressed | File Header Object |
| Section 1 | Uncompressed | T1 |
| Section 2 | Compressed | D |
| Section 3 | Uncompressed | T2 |
| Section 4 | Compressed | B E A |

Because multiple root-level objects are allowed in the file format, there is no obvious end point in the data. In order that the loader can determine that the file has ended, the total length of the file is stored in the header. Reading from the file is ended when the total number of bytes is reached. At this point, any objects not yet linked into the scene graph are treated as root-level objects and returned to the application.

## 8.1 Object References

Each object serialized, including the header object, is given an index, in order, starting from 1. The 0 index is used to indicate a null reference. This index is unrelated to the user ID for an object.

A reference to an object is serialized as an integer containing its index within the file. The serialization order constraint can be expressed as follows:

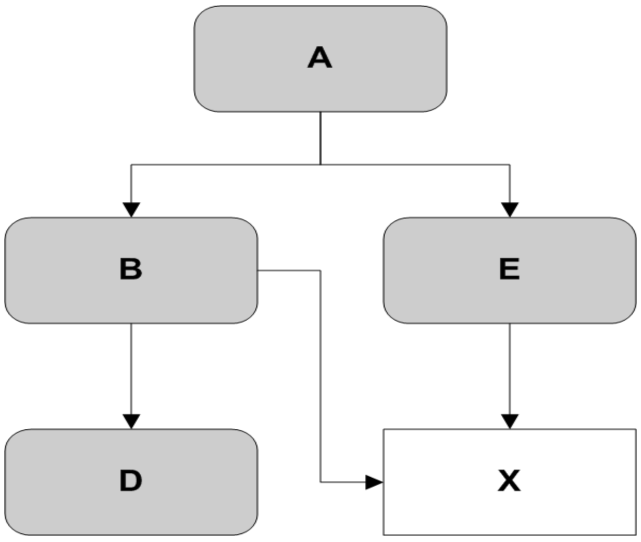
For an object with index **i**, a reference index **r** within it is only valid if **r** <= **i**. Invalid reference indices must be treated as an error.

An object reference must refer to an object of a valid type for the reference involved. For example, the reference from an Appearance object to a Material object must actually refer to a Material object. If the referred object type is incorrect, this must be treated as an error.

## 8.2 Shared Objects

Shared objects are handled the same way as normal objects. We only need to make sure that a shared object is sent before both of the objects that reference it.

For instance, for the following tree of references, where X is a shared object



A possible ordering would be D X B E A. Both the leaves-first and the depth-first algorithms described above will generate valid orderings.

### 8.3 Subtree Loading

With reference based order, it will be more difficult to load an arbitrary subtree from a file than with, e.g., root-first order. However, it is still possible, using two passes of the file. Assume that the subtree is defined as "object number X and all its children". In the first pass, only the node references are decoded, and an empty tree of the complete scene graph is created. From this graph, we find node X and make a list of all its descendants in the subtree. During the second pass, we simply load all the objects in the list. The last object that was loaded will be the root of the subtree.

In the case where rewinding the stream is not possible, it is also possible to do subtree loading in just one pass. This is achieved by loading everything up until the root node of the desired subtree, and then letting the garbage collection remove everything that is not referred to by the subtree. However, such an implementation would consume more memory than the two-pass implementation above. In the worst case, this is no different from loading the entire scene. For example, if the file contains a 3D map of the whole world and all you want is a specific house, you may still need to load the entire world and then delete everything but the house, if the house is the last object in the file.

# 9 Error Handling

There are several points at which an error may be detected. These include, but are not limited to:

- Memory exhaustion
- Missing or malformed file identifier
- Invalid section type
- Invalid file, section, or object length
- Invalid section checksum
- Invalid object type
- Extra or missing object data
- Invalid object reference
- Invalid enumeration value
- Invalid boolean value
- Invalid floating point value
- Values out of range for property
- Attempt to read past end of stream
- Aborted download
- Error in external reference

In particular, if values read from the loaded file would cause an *immediate* exception when passed to the API (e.g. all the attenuation parameters on a light are 0.0, or an image is too large for the implementation to handle), then this must be treated as an error.

If combinations of values are read that may cause a *deferred* exception, (e.g. a material and light are both present, but there are no normals specified), then this must *not* be treated as an error by the Loader. The application must be given the opportunity to take action after loading, in order to avoid these exceptions.

If any kind of error is detected during loading, the required action is for the Loader to abort this download, and that of any pending external references, clear up any temporary data structures, and throw an exception. If this file is being used as an external reference, then this is also treated as an error in the file that is attempting to load it. (This definition is, of course, recursive.)

The practical upshot of this is that any error detected in any of the files that may make up a world being loaded must result in a safe abort of the loading process and the throwing of a single exception to the main application.

It is up to the application what action, if any, is taken in the event of a loading error. Options range from an apologetic alert to the user ("Download failed!"), up through sophisticated error recovery schemes involving alternate file locations, or even different content in extreme cases.

# 10 Special Object Data

The data for the "special" object types is documented here.

## 10.1 Header Object

Object Type: 00
Superclass data: *none*
Followed by:

```
Byte[2] VersionNumber
Boolean hasExternalReferences
UInt32  TotalFileSize
UInt32  ApproximateContentSize
String  AuthoringField
```

There must be exactly one Header object in a file, and it must be the only object in the first section of the file, which in turn must be uncompressed. Due to its position in the file, it will always be assigned object index 1.

`VersionNumber` is a unique version number identifying variants of the file fomat. Only one variant is currently specified: version number 1.0. This must be indicated by `VersionNumber = {1, 0}`. The first number is major revision number, followed by minor revision.

`hasExternalReferences` is a boolean that describes whether this file is self-contained or includes URIs for other files, such as textures or geometries. If this is `false`, the file is self-contained. If it is `true`, then it indicates that the immediately following section of the file will contain the external reference objects needed to specify these external links. See Section 10.2 for more details.

`TotalFileSize` is the total size of the file, from the start to the end. It will be used in the loading, so it must be correct. (That is, it is not a hint.) For example, a file of size 6783 would define this field as `TotalFileSize = 6783`.

`ApproximateContentSize` contains the total number of bytes required to dowload the entire scene, including external links such as textures and geometry. This is provided as a hint, so that the user can know how much data he/she will pay for before loading the entire scene. The `ApproximateContentSize` field is also necessary in order to produce a good progress bar during the loading of the scene.

It should be noted that this information is only a hint. For instance, the file sizes of the objects that this file is linked to might have changed. Due to this, it is called "approximate" content size. Note that `ApproximateContentSize` should be equal to `TotalFileSize` if `ExternalFiles` is false.

For example, a file of 6083 bytes, with an external reference to another file of 10700 bytes would set `ApproximateContentSize = 16783`.

`AuthoringField` consists of a single nul-terminated UTF-8 string. The content of the string is not defined, and may include any information that the authoring environment wishes to place into it. Its most common purpose is mainly to make it possible to put a copyright note on the file, for example: `AuthoringField = "Blast4Fun (C) 2003 Extreme Games Inc."`

Note that if the string just contains numbers and letters from the English alphabet, the UTF-8 encoding will be the same as ASCII encoding.

## 10.2 External Reference

ObjectType: 0xFF (255)
Superclass data: *none*
Followed by:

```
String  URI
```

Instead of storing an object in-place, it is possible to have an external reference, in the form of a URI. This is stored in the object data as a single, nul-terminated UTF-8 string.

Relative URIs are relative to the file in which they are found, as usual. For example, a URI of "http://www.gamesforfun.com/objs/redcar.m3g", indicates another file in the M3G file format, at an absolute address, and "bluecartexture.png" indicates a PNG file in the same location as the current file.

If an external reference cannot be loaded, this will result in an error, causing the parent file to be "unloadable".

Loops of external references (e.g. file A references file B which in turn references file A again) are illegal and will result in a loading error.

The loader must only indicate that the loading of a file is complete when all external references within it have also been successfully loaded and the references type checked.

External references may appear only within their own section within the file. If present, this appears immediately after the file header section. It may be compressed or uncompressed.

In order to facilitate type checking of external references, loading of the externally referenced file must complete before reading any objects which could refer to it. This is one of the main reasons for ensuring that external references are in their own section, which occurs before sections containing objects of other types. For example, if the external reference is referred to as if it were an Appearance object, then the check that it is indeed an Appearance can occur only after loading the referred file.

External reference loading must support both M3G and PNG file types in order to satisfy the specification. An external reference to any other type of file must be treated as an error. To stress the point, even if a particular format (e.g., JPEG) is otherwise supported by the Loader, it must still reject any M3G files that reference JPEG images.

For M3G format files, the external reference must be able to load another M3G format file containing a single root-level object. If more than one root-level object is defined, then the first root-level object will be used, and the other objects and their descendants discarded.

For PNG format files, the external reference must be able to reference a valid PNG file, in which case the object created is a single instance of Image2D.

In all cases, once loading completes, the single root-level object loaded from the file effectively replaces the external reference object in the object index table. References to that index will then nominate the root-level object.

# 11 Per-Class Data

The data for each class in the API is now presented in alphabetical order. Where a class is a subclass, the superclass's data is always output first, and this information is taken to be part of the data for the class as a whole.

Classes without a serialized form (e.g. Graphics3D) are shown here for completeness, but are indicated as "not a serializable class".

Detailed information about each field is not given - it should be assumed that the data have the same meanings as those assigned in the API. Where data is serialized in a form which is different from the way it is specified in the API, this alternate form is documented here.

Any values which would be invalid as arguments to the corresponding methods in the API are also invalid in the file and must be reported as errors. For example, a negative value in the light attenuation fields is disallowed by the API and is therefore also disallowed in the file format.

## 11.1 AnimationController

ObjectType: 01
Superclass data: Object3D
Followed by:

```
Float32       speed;
Float32       weight;
Int32         activeIntervalStart;
Int32         activeIntervalEnd;
Float32       referenceSequenceTime;
Int32         referenceWorldTime;
```

## 11.2 AnimationTrack

ObjectType: 02

Superclass data: Object3D
Followed by:

```
ObjectIndex   keyframeSequence;
ObjectIndex   animationController;
UInt32        propertyID;
```

The `propertyID` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.3 Appearance

ObjectType: 03
Superclass data: Object3D
Followed by:

```
Byte          layer;
ObjectIndex   compositingMode;
ObjectIndex   fog;
ObjectIndex   polygonMode;
ObjectIndex   material;
ObjectIndex[] textures;
```

These are simply references to each of the objects aggregated together to form an appearance.

There are as many texture objects in the textures array as there are active texture units for this appearance. The texture units are loaded sequentially from unit 0. If the implementation supports more texture units than are specified, these are left in their default, inactive state, with a null texture.

If more textures are specified than are supported by the implementation, then this must be treated as an error, as it would be in the API. The application can then decide on an appropriate course of action to handle this case.

## 11.4 Background

ObjectType: 04
Superclass data: Object3D
Followed by:

```
ColorRGBA     backgroundColor;
ObjectIndex   backgroundImage;
Byte          backgroundImageModeX;
Byte          backgroundImageModeY;
Int32         cropX;
Int32         cropY;
Int32         cropWidth;
Int32         cropHeight;
Boolean       depthClearEnabled;
Boolean       colorClearEnabled;
```

The `backgroundImageModeX` and `backgroundImageModeY` fields must each hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.5 Camera

ObjectType: 05
Superclass data: Node
Followed by:

```
Byte          projectionType;
IF projectionType==GENERIC, THEN
    Matrix        projectionMatrix;
ELSE
    Float32       fovy;
    Float32       AspectRatio;
    Float32       near;
    Float32       far;
END
```

The `projectionType` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.6 CompositingMode

ObjectType: 06
Superclass data: Object3D
Followed by:

```
Boolean       depthTestEnabled;
Boolean       depthWriteEnabled;
Boolean       colorWriteEnabled;
Boolean       alphaWriteEnabled;
Byte          blending;
Byte          alphaThreshold;
Float32       depthOffsetFactor;
Float32       depthOffsetUnits;
```

The `blending` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

The `alphaThreshold` field is stored as a byte to save space. It is mapped so that 0x00 is equivalent to 0.0 (completely transparent), and 0xFF is equivalent to 1.0 (completely opaque).

## 11.7 Fog

ObjectType: 07
Superclass data: Object3D
Followed by:

```
ColorRGB      color;
Byte          mode;
IF mode==EXPONENTIAL, THEN
```

```
        Float32       density;
    ELSE IF mode==LINEAR, THEN
        Float32       near;
        Float32       far;
    END
```

The `mode` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.8 Graphics3D

*Not a serializable class.*

## 11.9 Group

ObjectType: 09
Superclass data: Node
Followed by:

```
    ObjectIndex[] children;
```

## 11.10 Image2D

ObjectType: 10
Superclass data: Object3D
Followed by:

```
    Byte          format;
    Boolean       isMutable;
    UInt32        width;
    UInt32        height;
    IF isMutable==false, THEN
        Byte[]        palette;
        Byte[]        pixels;

    END
```

The `format` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

For a palettised format, the `pixels` array contains a single palette index per pixel, and the `palette` array will contain up to 256 entries, each consisting of a pixel specifier appropriate to the format chosen.

For a non-palettised format, the `palette` array will be empty, and the `pixels` array contains a pixel specifier appropriate to the format chosen.

In a pixel specifier, each byte is scaled such that 0 represents the value 0.0 and 255 represents the value 1.0. The different formats require different data to be serialized, as follows:

- ALPHA: a single byte per pixel, representing pixel opacity.
- LUMINANCE: a single byte per pixel, representing pixel luminance.
- LUMINANCE_ALPHA: two bytes per pixel. The first represents luminance, the second alpha.
- RGB: three bytes per pixel, representing red, green and blue respectively.
- RGBA: four bytes per pixel, representing red, green, blue and alpha respectively.

The meaning of the components is given in the documentation for the Image2D class.

## 11.11 IndexBuffer

ObjectType: *none (abstract base class)*
Superclass data: Object3D
Followed by: *no data (abstract class)*

## 11.12 KeyframeSequence

ObjectType: 19
Superclass data: Object3D
Followed by:

```
Byte          interpolation;
Byte          repeatMode;
Byte          encoding;
UInt32        duration;
UInt32        validRangeFirst;
UInt32        validRangeLast;
UInt32        componentCount;
UInt32        keyframeCount;
IF encoding == 0
    FOR each key frame...
        UInt32                  time;
        Float32[componentCount] vectorValue;
    END
ELSE IF encoding == 1
    Float32[componentCount] vectorBias;
    Float32[componentCount] vectorScale;
    FOR each key frame...
        UInt32               time;
        Byte[componentCount] vectorValue;
    END
ELSE IF encoding == 2
    Float32[componentCount] vectorBias;
    Float32[componentCount] vectorScale;
    FOR each key frame...
        UInt32                 time;
        UInt16[componentCount] vectorValue;
    END
END
```

The `interpolation` and `repeatMode` fields must each hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

All of the `vectorValue` arrays are the same size, so a separate count is stored outside the individual keyframe's data rather than with each array.

The `encoding` field indicates the encoding scheme to be used for the keyframe data. Only the nominated values above are allowed. Other values must be treated as errors.

- Encoding 0 indicates that the values are stored "raw" as floats.
- Encodings 1 and 2 indicate that the values are quantized to 1 or 2 bytes. For each component, a bias and scale are calculated from the sequence of values for that component. The bias is the mimimum value, the scale is the maximum value minus the minimum value. The raw values are then converted to a value 0..1 by subtracting the bias and dividing by the scale. These raw values are then quantized into the range of a Byte or UInt16 by multiplying by 255 or 65535 respectively. The converse operation restores the original value from the quantized values.

## 11.13 Light

ObjectType: 12
Superclass data: Node
Followed by:

```
      Float32       attenuationConstant;
      Float32       attenuationLinear;
      Float32       attenuationQuadratic;
      ColorRGB      color;
      Byte          mode;
      Float32       intensity;
      Float32       spotAngle;
      Float32       spotExponent;
```

The `mode` field must hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.14 Loader

*Not a serializable class.*

## 11.15 Material

ObjectType: 13
Superclass data: Object3D
Followed by:

```
      ColorRGB      ambientColor;
      ColorRGBA     diffuseColor;
      ColorRGB      emissiveColor;
      ColorRGB      specularColor;
      Float32       shininess;
      Boolean       vertexColorTrackingEnabled;
```

## 11.16 Mesh

ObjectType: 14
Superclass data: Node
Followed by:

```
ObjectIndex   vertexBuffer;
UInt32        submeshCount;
FOR each submesh...
    ObjectIndex   indexBuffer;
    ObjectIndex   appearance;
END
```

## 11.17 MorphingMesh

ObjectType: 15
Superclass data: Mesh
Followed by:

```
UInt32        morphTargetCount;
FOR each target buffer...
    ObjectIndex   morphTarget;
    Float32       initialWeight;
END
```

## 11.18 Node

ObjectType: *none (abstract base class)*
Superclass data: Transformable
Followed by:

```
Boolean       enableRendering;
Boolean       enablePicking;
Byte          alphaFactor;
UInt32        scope;
Boolean       hasAlignment;
IF hasAlignment==TRUE, THEN
      Byte          zTarget;
      Byte          yTarget;
      ObjectIndex   zReference;
      ObjectIndex   yReference;
END
```

The `zTarget` and `yTarget` fields must each hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

The `alphaFactor` field is stored as a byte to save space. It is mapped so that 0x00 is equivalent to 0.0 (fully transparent), and 255 is equivalent to 1.0 (fully opaque).

If the `hasAlignment` field is `false`, the omitted fields are initialized to their default values.

## 11.19 Object3D

ObjectType: *none (abstract base class)*
Superclass data: *none*
Followed by:

```
UInt32           userID;
ObjectIndex[]    animationTracks;
UInt32           userParameterCount;
FOR each user parameter...
    UInt32        parameterID;
    Byte[]        parameterValue;
END
```

The userID field may be any value.

The user parameter data contains enough data to create a `java.util.Hashtable` object. This contains key/value pairs, with the key being the `parameterID`, and the value being the `parameterValue` byte array. The meanings of the IDs, and the contents of the byte arrays, are defined by the application and may have any value.

The behaviour of the `java.util.Hashtable` class does not allow multiple objects with the same key. Therefore, duplicate `parameterID` values are not allowed and must be reported as an error.

If an object has no user parameters, the `userParameterCount` field must be 0. In this case, the user object in the resulting Object3D instance must be set to `null`, rather than indicating a Hashtable object with no content. The Hashtable containing the parameters, if it exists, can be retrieved through the API using the getUserObject method.

## 11.20 PolygonMode

ObjectType: 08
Superclass data: Object3D
Followed by:

```
Byte          culling;
Byte          shading;
Byte          winding;
Boolean       twoSidedLightingEnabled;
Boolean       localCameraLightingEnabled;
Boolean       perspectiveCorrectionEnabled;
```

The `culling`, `shading` and `winding` fields must each hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.21 RayIntersection

*Not a serializable class.*

## 11.22 SkinnedMesh

ObjectType: 16
Superclass data: Mesh
Followed by:

```
ObjectIndex   skeleton;
UInt32        transformReferenceCount;
FOR each bone reference...
    ObjectIndex   transformNode;
    UInt32        firstVertex;
    UInt32        vertexCount;
    Int32         weight;
END
```

## 11.23 Sprite

ObjectType: 18
Superclass data: Node
Followed by:

```
ObjectIndex   image;
ObjectIndex   appearance;
Boolean       isScaled;
Int32         cropX;
Int32         cropY;
Int32         cropWidth;
Int32         cropHeight;
```

## 11.24 Texture2D

ObjectType: 17
Superclass data: Transformable
Followed by:

```
ObjectIndex   image;
ColorRGB      blendColor;
Byte          blending;
Byte          wrappingS;
Byte          wrappingT;
Byte          levelFilter;
Byte          imageFilter;
```

The `levelFilter`, `imageFilter`, `wrappingS`, `wrappingT`, and `blending` fields must each hold a valid enumerated value, as specified in the class definition. Other values must be treated as errors.

## 11.25 Transform

*Not a serializable class.*

## 11.26 Transformable

ObjectType: *none (abstract base class)*
Superclass data: Object3D
Followed by:

```
Boolean       hasComponentTransform;
IF hasComponentTransform==TRUE, THEN
    Vector3D      translation;
    Vector3D      scale;
    Float32       orientationAngle;
    Vector3D      orientationAxis;
END
Boolean       hasGeneralTransform;
IF hasGeneralTransform==TRUE, THEN
    Matrix        transform;
END
```

If either `hasComponentTransform` or `hasGeneralTransform` is `false`, the omitted fields will be initialized to their default values (equivalent to an identity transform in both cases).

## 11.27 TriangleStripArray

ObjectType: 11
Superclass data: IndexBuffer
Followed by:

```
Byte        encoding;
IF encoding == 0, THEN
    UInt32        startIndex;
ELSE IF encoding == 1, THEN
    Byte          startIndex;
ELSE IF encoding == 2, THEN
    UInt16        startIndex;
ELSE IF encoding == 128, THEN
    UInt32[]      indices;
ELSE IF encoding == 129, THEN
    Byte[]        indices;
ELSE IF encoding == 130, THEN
    UInt16[]      indices;
END
UInt32[]      stripLengths;
```

Bit 7 of the `encoding` field is equivalent to the `explicit` property on the index buffer, and will be 1 if the index buffer was constructed with explicit indices, or 0 if constructed with implicit indices. The other bits indicate the width of each index field. 0 indicates that the "raw" integer values are written, 1 indicates that a single byte will suffice, and 2 indicates that a 16 bit integer is sufficient to hold all the given index values. Values for the `encoding` field other than those explicitly nominated above are not allowed and must be treated as errors.

## 11.28 VertexArray

ObjectType: 20

Superclass data: Object3D
Followed by:

```
Byte          componentSize;
Byte          componentCount;
Byte          encoding;
UInt16        vertexCount;
FOR each vertex...
    IF componentSize==1, THEN
        IF encoding==0, THEN
              Byte[componentCount] components;
        ELSE IF encoding==1, THEN
              Byte[componentCount] componentDeltas;
        END
    ELSE
        IF encoding==0, THEN
              Int16[componentCount] components;
        ELSE IF encoding==1, THEN
              Int16[componentCount] componentDeltas;
        END
    END
END
```

The `componentSize` and `componentCount` fields must each hold a valid value, as specified in the constructor definition. Other values must be treated as errors.

The `encoding` field indicates the encoding scheme to be used for the keyframe data. Only the nominated values above are allowed. Other values must be treated as errors.

- Encoding 0 indicates that the values are stored "raw" as bytes or 16 bit integers.
- Encoding 1 indicates that the values are stored as differences from the previous value. Each component is treated separately, so that the difference is taken from the corresponding component in the previous vertex. For the first vertex, the previous value is taken to be 0. Decoding proceeds by initializing an accumulator to 0 for each component, and adding each value to the accumulator. In order that the deltas can be represented within the same number of bits as the raw values, the accumulators should be the same length as the values required (i.e. 8 or 16 bites) and be allowed to overflow. This also means that the accumulation is not dependent on the signed or unsigned nature of the deltas. (For example, the 8-bit sequence 0, 127, 126 can equally well be represented using deltas of 0, 127, -1 or 0, 127, 255.)

## 11.29 VertexBuffer

ObjectType: 21
Superclass data: Object3D
Followed by:

```
ColorRGBA     defaultColor;
ObjectIndex   positions;
Float32[3]    positionBias;
Float32       positionScale;
ObjectIndex   normals;
ObjectIndex   colors;
```

```
UInt32        texcoordArrayCount;
FOR each texture coordinate array...
       ObjectIndex   texCoords;
       Float32[3]    texCoordBias;
       Float32       texCoordScale;
END
```

If a texture coordinate array has only two components, the corresponding `texCoordBias[2]` element must be 0.0.

Null texture coordinate arrays are never serialized, regardless of their position. A single texture coordinate array will therefore always be serialized as belonging to texturing unit 0, regardless of its original unit it was assigned to.

There are as many references in the texture coordinates array as there are active texture units for this geometry. The texture coordinate references are loaded sequentially from texture unit 0. If the implementation supports more texture units than are specified, these are left in their default, inactive state, with a null texture coordinate reference and an undefined bias and scale.

If more texture coordinate references are specified than are supported by the implementation, then this must be treated as an error, as it would be in the API. The application can then decide on an appropriate course of action to handle this case.

### 11.30 World

ObjectType: 22
Superclass data: Group
Followed by:

```
ObjectIndex   activeCamera;
ObjectIndex   background;
```

# 12 ObjectType Values

This list shows what object type a specific ObjectType value maps to.

| ObjectType value | Object Type |
|---|---|
| 00 | Header Object |
| 01 | AnimationController |
| 02 | AnimationTrack |
| 03 | Appearance |
| 04 | Background |
| 05 | Camera |
| 06 | CompositingMode |
| 07 | Fog |
| 08 | PolygonMode |

| | |
|---|---|
| 09 | Group |
| 10 | Image2D |
| 11 | TriangleStripArray |
| 12 | Light |
| 13 | Material |
| 14 | Mesh |
| 15 | MorphingMesh |
| 16 | SkinnedMesh |
| 17 | Texture2D |
| 18 | Sprite |
| 19 | KeyframeSequence |
| 20 | VertexArray |
| 21 | VertexBuffer |
| 22 | World |
| 23 ... 254 | Reserved for use in future versions of the file format |
| 255 | External Reference |

Note that Object3D, Transformable, Node, and IndexBuffer are abstract classes and cannot be instantiated directly. They therefore do not appear in this list.