The GNU 3DKit Programming Guide

An official GNU project

Version 0.4 - August 11, 2002

Philippe C.D. Robert – probert@siggraph.org

Contents

List of Figures

Chapter 1

About This Guide

Welcome to the GNU 3DKit application development environment. The GNU 3DKit provides an object oriented application development framework for creating high performance 3D graphics applications. It is officially part of the GNU project as an extension to GNUstep, which provides an object oriented application development framework and tool set for use on a wide variety of computer platforms. GNUstep is based on the original OpenStep specification provided by NeXT, Inc. (now Apple).

The GNU 3DKit's main application is to render 3D graphics in real-time on commodity hardware, using a scene graph architecture. It interfaces the industry standard OpenGL graphics library to achieve this goal.

1.1 Why Use The GNU 3DKit?

The GNU 3DKit facilitates the design and implementation of complexe, high performance 3D graphics application in the Objective-C language. The kit provides advanced features and techniques to achieve high performance rendering of large data sets, it furthermore integrates tightly with the GNUstep and Cocoa application environments and automatically uses hardware accelerated graphics functionality, if available.

Some of the features wich are supported by the kit are listed below:

- High performance geometry rendering
- Efficient graphics state handling
- Sophisticated lighting and shadowing
- Flexible texture handling
- Tight integration with the GNUstep and Cocoa environments
- Native support for multithreading
- Level of detail handling

- View frustum culling
- Advanced visual effects (motion blur, full scene antialiasing, ...)
- Support for multiple backends (ie. OpenStep, SDL or GLX)

The GNU 3DKit is furthermore a true cross platform solution – it is available for most X11 based Unix systems[1] as well as for Mac OS X!

A Microsoft Windows version of the GNU 3DKit is currently not planned. Once GNUstep is available on that platform this might change, of course.

1.2 The Frameworks

The GNU 3DKit consists of 2 main frameworks as shown in Table 1.1. Additionally there a numerous backend libraries which interface specific, low level technologies, such as SDL, GLX or glut, used by the GNU 3DKit to provide event handling and displaying capabilities.

Table 1.1: The GNU 3DKit components

GeometryKit.framework	GeometryKit.h	Mathematical functions
RenderKit.framework	RenderKit.h	Main rendering framework
SDLKit	SDLKit.h	SDL backend
GLXKit	GLXKit.h	GLX X11 backend
GlutKit	GlutKit.h	GLX X11 backend

1.3 Where To Find Resources

More information and last minute changes can be found on the internet:

- *http://savannah.gnu.org/projects/gnu3dkit/*
- *http://www.fsf.org/software/gnu3dkit/gnu3dkit.html*

Information about GNUstep can be found at *www.gnustep.org*.

1.4 System Prerequisites

To compile and use the GNU 3DKit on a X11 based Unix system, install the following prerequisites:

- The GNU Objective-C compiler
- The GNUstep libraries

[1] The GNU 3DKit is known to work on Linux, Solaris and FreeBSD.

- An OpenGL 1.2 compliant library

On Mac OS X the native compiler and *Cocoa frameworks* can to be used instead.

The additional utility kits, such as the GlutKit or the SDLKit, might have additional prerequisits. Please check the installation notes for a complete list.

Chapter 2

The RenderKit

This chapter gives an overview about the RenderKit framework of the GNU 3DKit. It explains how to write common GNU 3DKit applications and points out important details and concepts.

2.1 Setting Up the Environment

To initialise a GNU 3DKit application and render custom data only a few steps have to be performed:

1. Initialise the GNU 3DKit via the `G3DApplication`[1].
2. Initialise and configure all camera(s).
3. Create the scene either by loading it from a file or programmatically.
4. Assign the scene to the camera(s) in use.
5. Enter the render loop.

In the next few sections we will describe the possibilities and restrictions of the GNU 3DKit in greater details.

2.1.1 The Camera Abstraction

The design and implementation of the new camera class is a complete rework. The **G3DCamera** controls the rendering process by using various helper objects, such as a **G3DFrustum**, a **G3DContext**, a **G3DFog** and a special drawable object. The GNU 3DKit will support drawables for multiple programming interfaces:

- The NSOpenGLView class provided by Apple's Cocoa framework
- The Simple Direct Media Library (SDL)
- GLX on any X11 based system

[1] The **G3DApplication** singleton is the holder of all global information, such as the execution model, render backend and so on.

All of these implementations provide a unique way of accessing hardware level features. Their purpose is to encapsulate low level graphics and event handling functionality. The **G3DCamera** class thereby unifies the API in a backend neutral manner[2].

You can use several cameras simultaneously to render different views of the same scene within the same application. This is often used in CAD applications.

Setting Up the Viewport

The viewport is a camera attribute which can be accessed from the camera object via `-setViewport:` and `-viewport`. Changing the viewport values results in a modified drawable. The viewport is stored in a G3DViewport structure:

x, y: Specify the lower left corner of the viewport rectangle.

w,h: Specify the width and height of the viewport.

Setting Up the Frustum

The frustum is also a camera attribute, it defines the visible viewing volume of a concrete camera. A frustum is defined by

- the Field Of View (FOV)
- the *near* and *far* clipping planes

This information is managed by the camera's **G3DFrustum** object which internally uses a perspective matrix that produces a perspective projection as known from `glFrustum()`.

The GNU 3DKit camera furthermore supports a special pretransformation matrix which can be used to control off axis frustums.

Setting Up the Viewpoint

The viewpoint of a camera defines its position and orienation in world coordinates. It is set by calling ie. `-setPosition:orientation:up:` or by passing a view matrix to `-setMatrix:`.

2.1.2 Instantiating 3D Scenes

The GNU 3DKit manages scene data in a scene graph structure which facilitates several rendering optimisation techniques. Scene graphs can be created programmatically or by importing data from a file.

Importing Scene Data

Scene data can be imported from a file using the **G3DSceneManager** class. The GNU 3DKit does not define a file format of its own, instead specific file loaders are used to transform externally stored data into a GNU 3DKit scene.

To load scene data from a file and create a GNU 3DKit scene graph, the steps as as shown in Listing 2.1 have to be performed.

[2] See G3DCamera.h

```
- (G3DScene *)loadFile:(NSString*)path
{
    G3DSceneManager *mgr = [G3DSceneManager defaultManager];
    G3DScene *scene = [mgr sceneWithContentsOfFile:path];

    [scene prepareScene];

    return scene;
}
```

Figure 2.1: Importing a scene into the GNU 3DKit.

2.2 The Graph Nodes

A scene graph consists of a number of nodes arranged as an acyclic, directed graph. The nodes are implemented in a class hierarchy which is distinct from the graph hierarchy defined by a concrete instance of a scene graph.

All nodes of a GNU 3DKit scene graph are subclasses of the semi abstract **G3DGraphNode** class. Table 2.1 lists the most common node classes available in the GNU 3DKit.

Table 2.1: The GNU 3DKit graph nodes

G3DGraphNode	Abstract	Parent of all graph nodes
G3DScene	Root	Root of a scene
G3DGroup	Branch	Basic grouping node
G3DLODGroup	Branch	Level of detail grouping node
G3DSwitch	Branch	Switching group node
G3DTimedSwitch	Branch	Timed switch node
G3DShape	Leaf	Leaf providing geometric data
G3DLight	Leaf	Light source

2.3 Node Attributes

Many of the GNU 3DKit graph nodes make use of special attribute objects. Attributes are needed to exactly describe a node's state. Table 2.2 lists the most used attribute classes available in the GNU 3DKit.

2.4 The Rendering Loop

Once a scene graph is created, either by loading it from a file or programmatically, it can be rendered and processed in a variety of ways. This always happens by traversing the tree either preorder, inorder, postorder or any combination of these.

Table 2.2: The GNU 3DKit attributes

G3DColour	basic	Colour information stored as a 4-tuple
G3DState	advanced	Graphics state properties for shapes
G3DTexture	advanced	Advanced texture object
G3DGeometry	advanced	Geometry data holder

2.4.1 Execution Modes

Rendering of one frame is always split into multiple stages. Depending on the execution mode these stages can be processed entirely or partially in parallel. The execution mode is set via the **G3DApplication** object.

APP: The application traversal updates the dynamic elements in a scene and readies them for the next frame. It makes sure that the scene is in sync and all transformation matrices are correct.

CULL: The culling traversal selects the visible portions of a scene graph and puts them into a display list to be rendered by the DRAW stage.

DRAW: The draw traversal sends rendering command to the graphics hardware based on the content of the current display list.

To render a frame the camera's `-(void)render:` method has to be invoked. Be aware that depending on the execution mode this may result in a frame latency.

2.4.2 APP Traversal

The scene graph is always traversed in depth-first order. The APP traversal is initiated by calling `-(void)render:` on the camera. This traversal is executed once *per scene*.

The APP traversal updates dynamic elements in the scene, such as switch or LOD groups, or invokes custom code:

1. Prune the node based on the traversal mask, if required. If the node is pruned then proceed with its sibling.

2. Invoke the render delegate's pre traversal method, if specified. Based on its return value prune, continue or terminate traversal.

3. Traverse all existing children.

4. Invoke the render delegate's post traversal method, if specified.

Figure 2.2 lists all delegate methods related to the APP traversal phase.

```
- (BOOL)nodeShouldTraverse:(id)sender;
    // Called from a node before APP traversal is initiated.

- (void)nodeWillTraverse:(id)sender;
    // Called from a node before APP traversal is initiated.

- (void)nodeDidTraverse:(id)sender;
    // Called after the traversal has been processed.
```

Figure 2.2: The node APP delegate methods.

Bounding Volumes

The GNU 3DKit automatically computes bounding volumes for every node in a scene graph unless it is set by the programmer. Bounding volumes are always updated upon change[3] – this can be prevented by declaring them static.

Bounding volumes are created hierarchically. Based on the bounding volumes provided by the **G3DGeometry** objects, all the shape nodes create their *bounding spheres*, which in turn are used by their parents to compute their bounding volumes, and so on.

Bounding volumes for geometry sets are computed once they are added to a shape node or upon invocation of `- (void)update`.

Adding a geometry object to a shape node does not automatically trigger the updating of the shape's bounding sphere, this has to be done explicitly! Furthermore, upon a change of a geometry object the respective bounding box is **not** being updated transparently. This as well has to be done explicitly by the programmer.

2.4.3 CULL Traversal

The CULL traversal stage is used to determin the visible part of the scene for the current frame. Updated bounding volumes are required to perform this step correctly! Starting at the scene graph's root node all of its children are tested for visibility using the camera's viewing frustum[4]. It is done as follows:

1. Prune the node based on the draw traversal mask , if required.
2. Invoke the render delegate's pre cull traversal method, if specified. Based on its return value prune, continue or terminate traversal.
3. Prune the node if its bounding volume is entirely outside the viewing frustum.
4. If the bounding volume is only partially within the viewing frustum, traverse all of the node's children[5] or geometry objects. If the bounding

[3] I.e. upon applying a transformation operation.
[4] Specific occluders have to be selected programmatically if occlusion culling is enabled
[5] If it is a selective group node then traverse only the active child.

volume is entirely within the viewing frustum, mark all children as visible and continue with the node's sibling!

5. Invoke the render delegate's post cull traversal method, if specified.

If the CULL traversal should not test geometry objects for visibility this can be achieved by changing the traversal mode using `-(void)setTraversalMode:` on the rendering context[6]. Figure 2.3 lists all delegate methods which are called upon the CULL traversal.

```
- (BOOL)nodeShouldCull:(id)sender;
    // Called from a node before culling is initiated.

- (BOOL)nodeIsVisible:(id)sender;
    // Used to customise culling.
```

Figure 2.3: The node CULL delegate methods.

Display Lists

During the CULL traversal a `G3DDisplayList` is generated which holds the geometry and state information of all nodes to be rendered for the current frame. Such render lists are flattened in an optimal way and thus much faster to process than traversing the scene graph directly.

Recreation of such display lists can be prevented by compiling a certain part of the scene graph into a static list. This greatly enhances the overall rendering performance for static scenes.

2.4.4 Draw Traversal

The DRAW traversal is performed directly after culling took place. Thus its only purpose is to process the current display list and send the appropriate render commands to the graphics pipeline. Figure 2.4 lists all DRAW related delegate methods.

```
- (void)nodeWillDraw:(id)sender;
    // Called from a node before the DRAW traversal is initiated.

- (void)nodeDidDraw:(id)sender;
    // Called after the DRAW traversal has been processed.
```

Figure 2.4: The node DRAW delegate methods.

[6] The argument to this call is a set of logically ORed flags.

2.4.5 Render Delegates

Render delegates are used to implement functionality on scene graph nodes without subclassing them. Every node in a scene graph can be controlled or enhanced by such a delegate, as shown above for the different traversal delegate methods[7].

Render delegates can thus be used for many purposes, some examples are listed below:

- The CULL stage can be optimised by implementing customised culling methods via the cull delegate methods. This can for example be used to implement *occlusion culling* as described in [4].
- Upon camera movement *line of sight* information can be updated. This again can be used to use *occlusion culling*.
- Upon node position changes collision detection and intersection tests can be computed.
- Application status information can be modified upon node state changes notified via APP traversal delegates.

Using render delegates provides a flexible way to design a GNU 3DKit application by aggregating custom logic rather than by subclassing. This enhances the ease of maintenance and reusability of code.

2.5 User Data

Every node of a scene graph may hold user data. This provides some amount of flexibility to the application's writer to prevent subclassing of GNU 3DKit classes.

2.6 Frame Data

Frame data is passed to all nodes while rendering a frame via the camera's `-render:` method. Frame data is managed in a latency aware manner to guarantee proper results in a multithreaded environment.

2.7 Node Sharing

In many situations having shared instances of a node is not very useful[8]. There are some exceptions where it makes sense to share nodes, though:

- Geometric data is shared whenever possible. This reduces the overall memory consumption of the application.
- Textures are shared using OpenGL texture objects. This enhances the overall rendering performance quite a lot.

[7] See the class documenation for all scene graph nodes and the camera

[8] A shared instance is a node that has multiple parents in the scene graph.

Scene graphs which contain shared nodes can be *flattened* automatically via the `-(void)flatten:` method declared on `G3DGraphNode`. This may lead to a higher memory consumption but also to better frame rates.

For a more detailed introduction into flatting scene graphs read 5.2.1.

Chapter 3

The GeometryKit

In this chapter the features and functions of the GeometryKit are explained. The GeometryKit is extensively used by the RenderKit but can also be used by programmers to do mathematical computations.

3.1 Introduction

The design of the framework is heavily influenced by the concepts and paradigms of the OpenStep API. The emphasis of the GeometryKit is on providing a complete set of mathematical functionality. The biggest part of the GeometryKit's core implementation is written in highly optimised ANSI C. It can be accessed either directly, or by using the Objective-C wrapper classes. A lot of the mathematical codebase is based on the famous *Graphics Gems* - I suggest the mathematical inclined to have a look into all of those books, especially [2] and [19].

In order to remain platform independent no architecture specific assembler optimisations have been introduced. However, most of the C functions are implemented as *inline* functions to attain a maximum of performance. Everything you need to know about ANSI C is covered in [17].

All base classes and the entire ANSI C foundation of the GeometryKit have both single-precision and double-precision floating point value representations. To resemble the OpenGL notation scheme the following convention is used throughout the GNU 3DKit:

void G3DFunctionName[n]{ifd}{v}(arg1, arg2, ...);

The number n indicates the dimension of the expected arguments, while the prefix *G3D* is used exclusively by all functions and macros of the GNU 3DKit.

3.2 Vectors

Vectors are available as two, three or four element implementations. All **G3DVector*** classes implement basic operations common to all vectors, such as reading, setting and clamping elements to specified high and low values, computing the

scalar multiplication and division, addition and subtraction of other vectors, interpolation and negation and so on. Furthermore typical vector functionality such as the dot product, the length of a vector, the cross product (in three dimensions only!) as well as normalisation is also directly available. Figure 3.1 lists all the available vector implementations of the GNU 3DKit.

Table 3.1: The GNU 3DKit vector classes

G3DVector2f	float	2 tuple in float representation
G3DVector2d	double	2 tuple in double representation
G3DVector3f	float	3 tuple in float representation
G3DVector3d	double	3 tuple in double representation
G3DVector4f	float	4 tuple in float representation
G3DVector4d	double	4 tuple in double representation

Note that the vector classes are subclasses of their respective **G3DTuple*** classes from which they inherit the common tuple functionality.

3.3 Quaternions

Quaternions are extensions of complex numbers, often used to compute rotations and orientations in 3D graphics in an efficient way. See [2], [6] and [3] to learn more about the mathematical background and examples of quaternions. Quaternions are implemented by **G3DQuaternion*** classes as listed in Figure 3.2.

Table 3.2: The GNU 3DKit quaternion classes

G3DQuaternionf	float	quaternion in float representation
G3DQuaterniond	double	quaternion in double representation

Note that the quaternion classes are subclasses of their respective **G3DTuple*** classes from which they inherit the common tuple functionality.

3.4 Matrices

Three and four dimensional matrices are among the most often used geometric tools in 3D graphics. An optimised implementation is therefore the prerequisite to reach optimal performance. Matrices are implemented by **G3DMatrix*** classes as listed in Figure 3.3.

Table 3.3: The GNU 3DKit matrix classes

G3DMatrix3f	float	3x3 matrix in float representation
G3DMatrix3d	double	3x3 matrix in double representation
G3DMatrix4f	float	4x4 matrix in float representation
G3DMatrix4d	double	4x4 matrix in double representation

3.5 Geometric Primitives

In addition to the basic algebraic functionalities described above, the GeometryKit also offers some geometric primitives which are used quite often in 3D graphics applications, i.e. to compute collision detection, bounding volumes and so forth. These objects are high level constructs, based on the optimised C foundation in float representation.

3.5.1 Spheres

Spheres are implemented by the **G3DSphere** class, a common sphere object, implemented as an object wrapper of the sphere's radius and center point.

3.5.2 Boxes

Boxes are implemented by the **G3DBox** class, which are used by the GNU 3DKit to implement *axis aligned bounding boxes*[1].

3.5.3 Planes

A **G3DPlane** is an object wrapper based on the well known plane equation

$$a \cdot x + b \cdot y + c \cdot z = d$$

based on a single-precision floating point 4 tuple.

3.5.4 Lines

Lines are a mathematical line abstraction implemented by the **G3DLine** class as

$$L(t) = Origin + t \cdot Direction$$

where *Direction* is always normalised.

3.6 Higher Level Constructs

In addition to these common primitives, the GeometryKit also provides support for more advanced constructs.

[1] Often referred to as AABB.

3.6.1 Bezier Curves

The GNU 3DKit implements *Bezier curves* in the **G3DBezier** class, a generic bezier curve implementation using n control points.

3.6.2 NURBS

Not yet available.

3.7 Physical Computation

Not yet available.

Chapter 4

Advanced Features and Rendering Techniques

In this chapter some more advanced features of the GNU 3DKit will be explained, such as creating advanced graphics effects or aspects of simulations.

4.1 Synchronisation

For certain applications it is crucial to be able to control frame rates and rendering synchronisation very exactly.

4.1.1 Fixed Frame Rates

Not yet available.

4.2 Shading And Lighting

The RenderKit helps application implementors to achieve realistic rendering effects by wrapping commonly used OpenGL functionality in Objective-C classes. Nevertheless if direct access to OpenGL functionality is still required this can be done by following some general rules set by the GNU 3DKit.

4.3 Texturing

The GNU 3DKit provides a common texture class which takes care of loading textures from a file and controlling the OpenGL texture functionality. Multitexturing is provided as well, the GNU 3DKit makes thereby use of the multitexture extension, if available.

4.4 Environment Effects

The GNU 3DKit supports some global environment effects such as ie. fog directly.

4.4.1 Fog

Fog is enabled via the **G3DCamera** objcet and controlled by its **G3DFog** object.

4.5 Advanced Visual Effects

The GNU 3DKit furthermore supports advanced visual effects in order to achieve highly realistic rendering results.

4.5.1 Motion Blur

Motion blur can be enabled on the camera object. Motion blur requires the existence of a hardware accelerated accumulation buffer, though. Otherwise the rendering performance will drop dramatically!

4.5.2 Full Scene Antialiasing

Full Scene Antialiasing (FSAA) can be enabled on the camera object. FSAA requires the existence of a hardware accelerated accumulation buffer, though. Otherwise the rendering performance will drop dramatically!

4.5.3 Shadow Volumes

Not yet available.

4.5.4 Cell Shading

Not yet available.

Chapter 5

Tuning and Optimisations

In this final chapter we have a closer look at optimisation issues as well as potential bottlenecks for 3D applications, based on the GNU 3DKit.

5.1 Multithreading

In order to support SMP systems in an optimal way the GNU 3DKit offers multithreaded rendering modes. The rendering mode of choice has to be specified upon initialisation of the application, currently the following modes are supported:

- All stages are processed by the same (application) thread.
- The DRAW and the CULL stage are processed by a separate thread.
- The DRAW and the CULL stage are processed by separate threads.
- The DRAW stage is processed by a separate thread.

The execution mode has to be specified upon mandatory initialisation of the **G3DApplication** before any other GNU 3DKit functionality is used!

5.1.1 Synchronisation Primitives

The GNU 3DKit makes use of the elegant synchronisation primitives offered by the GNUstep API. Unless other scene graph APIs it therefore does not have to provide special synchronisation primitives of its own, this guarantees optimal integration with GNUstep and Cocoa.

5.2 Scene Graph Optimisations

In order to achieve the best performance it is crucial that the scene graph is organised in an optimal way. Only by minimising the required traversals and state switches in the graphics pipeline it is possible to get realtime results. In the next sections some of the most important issues are explained and shown how to deal with using the GNU 3DKit.

5.2.1 Flattening

One way to reduce traversal complexity is to flatten a scene graph.

5.2.2 Sorting

In order to reduce the required state changes of the OpenGL rendering pipeline, the nodes of the scene graph are sorted in an optimal order.

5.3 Scene Optimisations

A prerequisite to create an optimal scene graph is the underlying structure of the used database.

5.3.1 Spatial Organisation

Not yet available.

5.3.2 Complexity

Not yet available.

Bibliography

[1] Apple Developer Documentation, Object-Oriented Programming and the Objective-C Language. Apple Computer, Inc., 1999

[2] Arvo, J., Graphics Gems II. Academic Press, Inc., 1991

[3] Bobick, N., Rotating Objects Using Quaternions. Game Developer, Vol. 2: Issue 26, July 3, 1998

[4] Coorg, S., Teller, S., Real-time occlusion culling for models with large occluders. Proc. 1997 ACM Symposium, on interactive 3D graphics, pp. 83 - 90 and 189

[5] Course Notes, Advanced Graphics Techniques Using OpenGL. Course notes, 1998

[6] Course Notes, Using Quaternions to Represent Rotation. CS184, UC Berkeley

[7] Duchaineau, M. et al., ROAMing Terrain: Real-Time Optimally Adapting Meshes. http://www.llnl.gov/graphics/ROAM/

[8] Foley, D., van Dam, A., Feiner, S., Hughes, J., Computer Graphics - Principles and Practice. Addison Wesley Co., Inc., Reading, MA, 1996

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns. Addison Wesley Co., Inc., Reading, MA, 1995

[10] Garland, M., Heckbert, P., Surface simplification using quadric error metrics. Proceedings SIGGRAPH 97, pp. 209 - 216

[11] Glassner, A.S., Graphics Gems. Academic Press, Inc., 1990

[12] Green, N., Environment Mapping and Other Applications of World Projections. IEEE Computer Graphics and Applications, vol. 6, no. 11, pp. 21 - 29, November 1986

[13] Heckbert, P.S., Herf, M., Shadow Generation Algorithms. http://www.www.cs.cmu.edu/ ph/shadow.html, 1997

[14] Hoppe, H., Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. Microsoft, http://www.research.microsoft.com/ hoppe

[15] Hoppe, H., Progressive Meshes. Proc. SIGGRAPH 96, pp. 99 - 108

[16] Kempf, R. and Frazier, C., OpenGL Reference Manual. Addison Wesley Co., Inc., Reading, MA, 1999

[17] Kernighan, B.W. and Ritchie, D.M., The C Programming Language. Prentice Hall Software Series, Prentice-Hall, Inc., 2nd Edition, 1988

[18] Kilgard, M.J., Avoiding 16 Common OpenGL Pitfalls. Copyright 1998, 1999 by Mark J. Kilgard, http://www.opengl.org

[19] Kirk, D., Graphics Gems III. Academic Press, Inc., 1992

[20] Lindstrom, P. et al., Real-Time Continuous Level of Detail Rendering of Height Fields. Proceedings SIGGRAPH 96

[21] Loshin, D., Efficient Memory Programming. The MacGraw-Hill Companies, Inc., 1999

[22] Moeller, Th., Haines, E., Real-Time Rendering. A K Peters, Ltd., 1999

[23] Watt, A., Policarpo, F., The Computer Image. Addison Wesley Co., Inc., Reading, MA, 1998

[24] Watt, A., Policarpo, F., 3D Games - Real-time Rendering and Software Technology. Addison Wesley Co., Inc., Reading, MA, 2001

[25] Woo, M., Neider, K. and Davis, T., OpenGL Programming Guide. Addison Wesley Co., Inc., Reading, MA, 1998

[26] Wright, R., Understanding and Using OpenGL Texture Objects. Game Developer, July 23, 1999

[27] Zhang, H., Effective Occlusion Culling for the Interactive Display of Arbitrary Models. Department of Computer Science, University of North Carolina at Chapel Hill, July 1998

[28] Zohar, R., Barad, H., Implementing a 3D SIMD Geometry and Lighting Pipeline. Game Developer, April 1999