Universitaire Instelling Antwerpen
Department of Mathematics and Computer Science

Advanced Computer Graphics using OpenGL.

Sven Maerivoet

http://svengl.dyns.cx

2000 - 2001

Contents

List of Figures

Chapter 1

Introduction

In this manual, a general framework for easily creating OpenGL-applications is considered. This framework is written in C++ and relies heavily on the concept of Object-Oriented Programming (OOP). General knowledge of encapsulation, inheritance and polymorphism is assumed. Various aspects are discussed, ranging from a general introduction and detailed explanations of class-methods to an in-depth treatment of the ray tracer, which was (partially) built using this framework.

Much of the work is based on the book [Hil01], the OpenGL-standard as defined in [SA99], the GLUT-API as defined in [Kil96] and the 'standard-bible' [FvDFH90]. The programming-techniques used, are based on those as presented in [Str97] and [CL95].

There are also several interesting papers available : [3DC01], [ART01], [Arv86], [Bry00], [BHH00], [Bus00], [CSS], [CS], [CK01], [Cla01], [Cof01], [DH92], [Eli00], [FSJ], [Gan01], [GTS], [Hai01], [Har01], [Hun00], [IRT01], [Kaj83], [KA88], [KMH], [LKR+96], [MCFS00], [Met00], [NYH01], [Nor97], [PPD98], [PSL+98], [PMS+99], [Per97], [PKGH97], [Pix00], [RSH], [RJ97], [Sch], [SW92], [Sto95], [TCRS00], [The01], [Tig99], [Tur], [WSB], [WBWS01], [WS01] and [Zha00].

1.1 A word about the framework

The general idea was to use (and create) as little low-level code as possible and to construct a high-level interface. This resulted in the use of GLUT, the *OpenGL Utility Toolkit* and the creation of a class-interface, suited for most needs. It is therefore required to operate on a system which has OpenGL installed, as well as the availability of GLUT.

It is assumed that the software-package (as it is available) is installed in a default directory, called ∼**/SvenGL**[1]. Within this directory, several noteworthy subdirectories are located. One of them is the ∼**/SvenGL/data**-directory which contains over 20 Mb of texture-images, mesh-files

[1] Note that a UNIX/Linux-style convention is used for directories and filenames.

and SDL-scenes. Another important subdirectory is ∼**/SvenGL/docs**, in which several documentation files (all the key-assignments for the various applications and the syntax of the SDL-language) can be found. The most important subdirectory however, is ∼**/SvenGL/opengl**, which contains the library-files.

1.2 How to use the library

From the high-level point of view, the only thing an application-programmer needs to do, is to derive his own *GL-class* (this is the name used for denoting a class that performs OpenGL-graphics) from the base-class `TOpenGLApp`, and instantiate it in a file containing the proper initialization-code inside it's `main()`-function.

This scheme might seem a bit awkward at first, but it is the only logical solution to a rather complex problem : GLUT relies on callback-functions, which are actually pointers to C-functions passed as parameters to the GLUT-API. If however, a C++ class is used, these pointers no longer refer to C-functions but to C++ member-functions. This introduces some annoying difficulties, and the only solution that seems straightforward, is to declare callback-functions as virtual member-functions (so they can be overridden) and to call these member-functions from within file-scope C-functions (called *wrappers*). These C-functions are then passed to the GLUT-API. The whole process can be a automated to a degree, but it is necesarry to understand this scheme.

1.2.1 The GL-class

Typically, the file `"opengl/opengl.h"` is included. This gives access to the base-class, from which a specific GL-class is derived. For example :

```
// ----------------------------
// this file is called demo-gl.h
// ----------------------------

#include "opengl/opengl.h"

using namespace std;

class DemoGLApp : public TOpenGLApp {
  public:
    // constructor
    DemoGLApp(int aParameter);

    // destructor
    ~DemoGLApp(void);

    // explicit initialization
    void explicitInitialize(void);
```

```
  private:
    // data-members
    int fClassMember;
};
```

It is straightforward to create standard OpenGL-applications, just override the proper callback-functions (see section 2.1.4 for a detailed description of the supported callback-functions).

Passing parameters (for example, command-line parameters) to the derived GL-class, can be accomplished by using a specific constructor (as can be seen in the above example) with an initialization-list (see the example below). There is also a special method, called `initialize()`, that the base-class uses to perform some OpenGL-specific configurations. In order to perform specific initialization of GL-class specific data, the `explicitInitialize()`-function can be overridden (it is automatically called by `initialize()`).

```
// ------------------------------
// this file is called demo-gl.cpp
// ------------------------------

#ifdef _WIN32
  #include <windows.h>
#endif

#include "demo-gl.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

using namespace std;

// ---------------
// class DemoGLApp
// ---------------

// -----------
// constructor
// -----------
DemoGLApp::DemoGLApp(int aParameter) : fClassMember(aParameter)
{
}

// ----------
// destructor
// ----------
DemoGLApp::~DemoGLApp(void)
{
}
```

```
// -----------------------
// explicit initialization
// -----------------------
DemoGLApp::explicitInitialize(void)
{
  // setup camera-position and -shape
}
```

Note (1) how the OpenGL-library is included at the beginning of the file and (2) the use of the `#ifdef _WIN32` directive.

1.2.2 The instantiator

In section 1.2, the general underlying concept was explained. Some file-scope C-functions are necessitated, and these are typically included in a standard file (the *instantiator*) that also contains the `main()`-function. For example :

```
// --------------------------------
// this file is called demo-main.cpp
// --------------------------------

#include "demo-gl.h"
#include "opengl/opengl.h"

using namespace std;

// --------------------------------
// global OpenGL Application object
// --------------------------------
TOpenGLApp* fOpenGLApp;

// --------------------------------
// wrappers for callback-functions
// --------------------------------
void displayFuncWrapper(void)
  { fOpenGLApp->antiAliasedDisplayFunc(); }
void reshapeFuncWrapper(int width, int height)
  { fOpenGLApp->reshapeFunc(width,height); }
void keyboardFuncWrapper(unsigned char key, int x, int y)
  { fOpenGLApp->keyboardFunc(key,x,y); }
void specialFuncWrapper(int key, int x, int y)
  { fOpenGLApp->specialFunc(key,x,y); }
void mouseFuncWrapper(int button, int state, int x, int y)
  { fOpenGLApp->mouseFunc(button,state,x,y); }
void motionFuncWrapper(int x, int y)
  { fOpenGLApp->motionFunc(x,y); }
```

```
void passiveMotionFuncWrapper(int x, int y)
  { fOpenGLApp->passiveMotionFunc(x,y); }
void idleFuncWrapper(void)
  { fOpenGLApp->idleFunc(); }

void cleanUp(void)
{
  if (fOpenGLApp != 0) {
    delete fOpenGLApp;
  }
}

void main(int argc, char* argv[])
{
  // register the cleanup-routine
  atexit(&cleanUp);

  // perform command-line parsing
  // note that convertStringToInt() must be defined !
  int parameter = convertStringToInt(argv[1]);

  // create and initialize the OpenGL Application object
  fOpenGLApp = new DemoGLApp(parameter);
  fOpenGLApp->initialize(0,0,
                         640,480,
                         "Demo OpenGL-application",
                         TOpenGLApp::kEnableDoubleBuffering,
                         TOpenGLApp::kEnableDepthBuffer);

  // register the necessary callback-functions
  fOpenGLApp->registerDisplayFunc(displayFuncWrapper);
  fOpenGLApp->registerReshapeFunc(reshapeFuncWrapper);
  fOpenGLApp->registerKeyboardFunc(keyboardFuncWrapper);

  // activate the event handling
  fOpenGLApp->activateEventHandling();
}
```

It is important to note that this **demo-main.cpp** includes the header-file in which the `DemoGLApp`-class is declared. When creating and initializing the global `fOpenGLApp`-object, the derived `DemoGLApp`-class is used (to access it's specific constructor). From here on, it is also possible to specify which callback-functions will be registered and which won't.

The description of the parameters to the function `defaultInitialize()` can be found in section 2.1.2.

1.3 Compiling and linking

Creating OpenGL-applications is one thing, getting them running is another. The fact that the whole interface is based on GLUT, has the advantage that any operating system supporting OpenGL and GLUT will be able to run the software. In the next two sections, compiling and linking the software on two popular kinds of operating systems (UNIX/Linux and Windows 95/98/2000/NT/ME) is discussed.

1.3.1 UNIX and Linux

When working under Unix or Linux, it is assumed that the GNU C++ compiler is available. A **Makefile** is provided and compiling and linking the software as-is, is nothing more than performing a simple '*make*'. Note that it might be necessary, depending on the current system configuration, to change some predefined settings in the **Makefile** :

```
CC=g++
LINKFLAGS=-L/usr/lib/libglut.so.3.7.0 -lGL -lGLU -lglut
```

The above default settings assume that the compiler is called **g++** and that the GLUT-library resides in **/usr/lib/libglut.so.3.7.0**.

In order to compile and link self-made programs, the **Makefile** can easily be modified to perform these tasks :

1. first, create an entry in the *user's entries*-section for the CPP-file (don't forget to specify the **opengl.o**-relation), for example :

   ```
   demo-gl.o: demo-gl.h demo-gl.cpp opengl.o
     $(CC) -c demo-gl.cpp -o demo-gl.o
   ```

2. Next, create an entry (in the same section) for the file containing the wrappers and the `main()`-function :

   ```
   demo-main: demo-main.cpp demo-gl.o
     $(CC) $(LINKFLAGS) demo-main.cpp $(OPENGLFILES) demo-gl.o -o demo-main
   ```

3. Finally, create an entry in the *clean entries*-section :

   ```
   rm -f cube
   ```

Compiling and linking is thus done by executing the '*make demo-main*'-command, running the program is done by executing the '*./demo-main*'-command and cleaning everything is done by executing the '*make clean*'-command.

1.3.2 Windows 95/98/2000/NT/ME

When compiling and linking under the Windows 95/98/2000/NT/ME operating system, it is assumed that Microsoft Visual C++ 6.0 has been installed and configured correctly. The following two subsections explain how to get the software up and running when using the default project and when starting from scratch.

Using the default project

A default Microsoft Visual C++ 6.0 project workspace is available.
It is called **svengl-msvcpp6.zip** and it is configured to run *SDLViewer*. Take the following steps in order to use it :

1. unzip **svengl-msvcpp6.zip** to a directory (e.g., **C:**\). This will create **C:****SvenGL****SvenGL.dsp** and **C:****SvenGL****SvenGL.dsw**.
2. Unzip all SvenGL-files (**data**-directory and sourcecode) to **C:****SvenGL**.
3. Open the project workspace by double clicking on the **C:****SvenGL****SvenGL.dsw**-file.
4. Select *Build SvenGL.exe* from the *Build*-menu and press *CTRL-F5* to run the application.

Doing it from scratch

Starting from scratch, the following steps should be taken :

1. start Microsoft Visual C++ 6.0 and select *New* from the *File*-menu. Click on the *Projects*-tab in the following dialog-box and select *Win32 console application*. Don't forget to specify a *project name* (e.g., **SvenGL**) and a *location* (e.g., **C:**\). Note that the project name is appended to the location. Click *Ok* to execute the changes.
2. In the following dialog-box, select the radio-button *An empty project*, click *Finish* and then click *Ok*.
3. Unzip all SvenGL-files (**data**-directory and sourcecode) to **C:****SvenGL**.
4. In the newly created workspace, select *Add to project* from the *Project*-menu and then select *Files* from the submenu. Double-click the **opengl**-directory (which was unzipped in the previous step) and press *CTRL-A* to select all files. Click *Ok* to close the dialog-box.
5. Repeat the previous step to add the newly created class from section 1.2. These files are called **demo-gl.h**, **demo-gl.cpp** and **demo-main.cpp**.

6. Select *Settings*. . . from the *Project*-menu and make sure that *Category* is set to *General*. In the left-side of the dialog-box, click on the *SvenGL*-project, then click on the *Link*-tab at the right and add the following files **opengl32.lib**, **glu32.lib** and **glut32.lib** to the *Object/library modules*. Click *Ok* to close the dialog-box.

7. Don't forget to save the workspace by selecting *Save Workspace* from the *File*-menu.

8. Select *Build SvenGL.exe* from the *Build*-menu and press *CTRL-F5* to run the application.

Note that it is possible to disable the popping-up of the console-window, by taking the following steps :

1. select *Settings*. . . from the *Project*-menu. Click on the *Link*-tab at the right and make sure that *Category* is set to *Output*.

2. Change the *Entry-point symbol* to **mainCRTStartup**.

3. In the list of *Project Options*, change */subsystem:console* to **/subsystem:windows**.

4. Don't forget to save the workspace by selecting *Save Workspace* from the *File*-menu.

5. Select *Rebuild All* from the *Build*-menu and press *CTRL-F5* to run the application.

Chapter 2

TOpenGLApp base class

The `TOpenGLApp`-class contains all the basic functions for drawing with OpenGL, as well as complete camera- and projection-management. This chapter describes in detail the class-methods that are available when deriving from this base class.

2.1 General considerations

Understanding how the base class operates, is crucial for quickly and successfully developing OpenGL-applications. Because the base class is built around GLUT, many low-level code has been replaced with easily accessible high-level class-methods. A central role is these methods is played by the so-called *primitives*, which can be found in ∼**/SvenGL/opengl/primitives.h** : `Point2D`, `Point3D`, `Rectangle2D`, `Vector2D`, `Vector3D` and `Color`. Note that nearly all the data-members of these classes are based on C++ doubles.

Figure 2.1 shows which various software-components talk to each other.

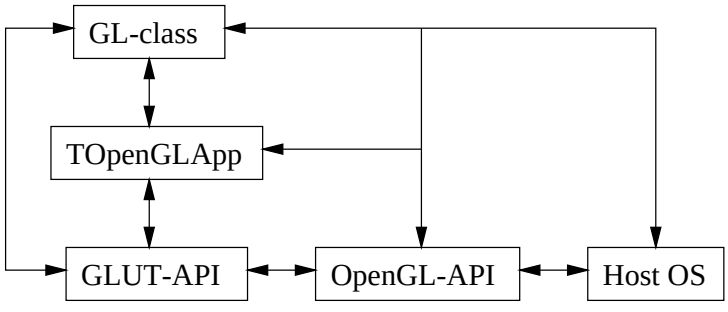

Figure 2.1: The various software-components.

2.1.1 Useful constants

Many constants are defined and because they are declared *static*, they can be accessed as follows :

```
TOpenGLApp::kQuasiFullScreen
```

These constants mainly come in two varieties : booleans and enums. They are declared to use meaningful names as parameters for the methods. In some of these methods – described further – these constants are used as parameters whenever appropriate.

2.1.2 Initialization

Besides the constructor of the derived class, there is also the following method which must explicitly be called, since it performs some OpenGL-specific configurations for the base class :

```
void initialize(int clientWindowXPos,
                int clientWindowYPos,
                int clientWindowXSize,
                int clientWindowYSize,
                string clientWindowTitle,
                bool enableDoubleBuffering,
                bool enableDepthBuffer);
```

In order to make an application-window that fills the entire screen, the `kQuasiFullScreen`-constant must be used for both the `clientWindowXSize`- and `clientWindowYSize`-parameters. Double buffering is necessary for flicker-free animations, the `kEnableDoubleBuffering`- and `kDisableDoubleBuffering`-constants are used to control this. The same holds for enabling and disabling the use of OpenGL's depth buffer, by using the `kEnableDepthBuffer`- and `kDisableDepthBuffer`-constants.

Explicit initialization of data-members in the derived class, can be accomplished by overriding the following method :

```
virtual void explicitInitialize(void);
```

Typically, the camera's position and shape are setup inside this method. Note that it automatically gets called by `initialize()`.

2.1.3 Callback registrations

The registration of the callback-functions to the GLUT-API is performed using the `registerXXXFunc()`-methods. These methods are to be called only once, during the initial registration-part, as shown in section 1.2.2. For optimal performance, only the needed callback-functions should be registered.

2.1.4 Default callbacks

As mentioned before, understanding how the base class works (and how GLUT works) is necessary for developing OpenGL-applications. The main scheme is as follows : all the needed callbacks are registered (each of them performs a specific task) and then GLUT's main-event-loop is activated.

The callbacks are divided into categories : screen management, user input and background processing. Note that not all GLUT-callbacks are supported (this was done in order to maintain a simple high-level interface).

Screen management

```
virtual void displayFunc(void);
virtual void reshapeFunc(int width, int height);
```

The default `displayFunc()`-callback shows a yellow cross on a blue background (this is visible when the callback is registered but *not* overridden). This callback is the most important and everything that has to be drawn should reside inside it. Note that the GLUT-API only redisplays a window when a redisplay is *posted* (e.g., when parts of it have changed), see section 2.1.6 for more details.

Note that the base class also contains an `antiAliasedDisplayFunc()`-callback which gets assigned as the default callback. This method must *not* be called directly, since it handles anti-aliasing by using OpenGL's accumulation-buffer in combination with a set of jitter-vectors. It automatically calls the (overridden) `displayFunc()`-callback.

The task of the default `reshapeFunc()`-callback is to maintain aspect-ratio of original world-window when the application's window is resized, so no distortions occur. The application's window's new dimensions are automatically given as the parameters `width` and `height` (both of them are expressed in pixels).

User input

```
virtual void keyboardFunc(unsigned char key, int x, int y);
virtual void specialFunc(int key, int x, int y);
virtual void mouseFunc(int button, int state, int x, int y);
virtual void motionFunc(int x, int y);
virtual void passiveMotionFunc(int x, int y);
```

The default callbacks for user input provide a rich set of controls, including camera and projection management, lighting, shading and other miscellaneous controls. The default behaviour is to simulate a controlled flight, by letting the camera move in the world-coordinate system, slide in its own coordinate system and rotate, pitch and yaw around its own axes. There is also a toggle provided for a windowed or a full-screen view, and support for taking snapshots of the visual

data in the application's window (these snapshots are saved to BMP-files, starting with the prefix **snapshot** and concatenated with a number, starting at 0, that counts the number of snapshots taken).

When overriding these functions, it is best that their counterparts in the base class still get called first, so the default behaviour and functionality is retained. A special measure has to be taken when overriding the `keyboardFunc()`-method, resulting in some 'redisplay-code' at the end of the method :

```
void keyboardFunc(unsigned char key, int x, int y)
{
  // call the counterpart in the base class
  TOpenGLApp::keyboardFunc(key,x,y);

  // perform specific key parsing
  // ...

  // post a redisplay to the GLUT-API
  if (fScheduleRedisplay) {
    redisplay();
  }
}
```

The `keyboardFunc()`-method is used for parsing ordinary keys (which are send as ASCII-characters), the `specialFunc()`-method is used for parsing function-keys (refer to the GLUT-API for more details).

When a user presses and releases mouse buttons, each press generates a mouse callback which can be handled by `mouseFunc()`. The `button`-parameter can be `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` or `GLUT_RIGHT_BUTTON`. The `state`-parameter can be `GLUT_UP` or `GLUT_DOWN`. The `x`- and `y`-parameters indicate – when the mouse button state changed – the mouse-position, relative to the application's window upper-left corner.

Capturing the movement of the mouse is accomplished using the `motionFunc()`- and `passiveMotionFunc()`-methods. The difference between them is that the first method gets called whenever the mouse moves within the application's window *and* one ore more mouse buttons are being pressed. The second methods gets called whenever the mouse moves within the application's window *and* none of the mouse buttons are being pressed. The `x`- and `y`-parameters indicate the mouse location in the application's window relative coordinates.

Note that there is *no* default behaviour for the mouse-related callbacks.

Background processing

```
virtual void idleFunc(void);
```

When window system events are not being received by the GLUT-API, the `idleFunc()`-method can be continuously called, which allows animation. The default behaviour is to enable continuous camera movements when flying (the flight can be controlled by the default behaviour the key-related callbacks offer) or momentum-mode (see section 2.1.5 for the distinction between the two modes).

2.1.5 Continuous camera movements versus momentum-mode

When the *continuous movement* of the camera is enabled, it will have the effect that each command given will continuously repeat. For example, pressing the key for rolling the camera to the left will let the camera roll until another command has been given (which then in turn will repeat itself) or until continuous movement is disabled.

On the other hand, *momentum-mode* will make the camera fly continuously in its current direction. It is however still possible to control this direction by rotating the camera around any of its three axes.

Note that both modes are mutually exclusive, which means that only one of them can be active at any time.

2.1.6 Miscellaneous functions

```
void enableAntiAliasing(double jitterFactor,
                        EJitterVectors nrOfJitterVectors);
void disableAntiAliasing(void);
void redisplay(void) const;
int getClientWindowXSize(void) const;
int getClientWindowYSize(void) const;
void takeSnapshot(string filename) const;
```

The use of anti-aliasing is toggled with the `enableAntiAliasing()`- and `disableAntiAliasing()`-methods. Anti-aliasing is done by slightly moving the camera around (controlled by means of the `jitterFactor`-parameter), and averaging all the obtained images which generally results in smoother transitions on the screen (as for example can be seen when OpenGL is scan-converting the primitives on the screen). The possible values for the `nrOfJitterVectors`-parameter, are : `kJitterVectors2`, `kJitterVectors3`, `kJitterVectors4`, `kJitterVectors8`, `kJitterVectors15`, `kJitterVectors24` and `kJitterVectors66`. Note that using more jitter-vectors results on the one hand in a higher image-quality, but on the other hand it *dramatically* slows down the rendering.

A word of caution is to be said about the `redisplay()`-method : *don't call this method from inside a `displayFunc()`- or `reshapeFunc()`-override !* It is however necessary to call it at the end of the other callback-functions, in order to post a redisplay to the base class, which in turn will post a redisplay to the GLUT-API (whilst taking into account the anti-aliasing that might need to be performed).

Refer to the user input in section 2.1.4 for the operation of the `takeSnapshot()`-method.

2.2 Viewing

The base class `TOpenGLApp` provides a rich set of methods for viewing two- and three-dimensional worlds. The window-to-viewport mapping basically handles two-dimensional drawing (be it a projection on the XY-plane).

2.2.1 Window-to-viewport mapping

Two main ingredients are needed : a window on the world and a viewport in the application's window. The viewport is directly related to the pixels on the screen, whilst the world-window is a completely different thing. A correspondence mapping has to be set up between the two. The only thing needed is the specification of the window and the viewport (the mapping is done automatically).

```
void setWorldWindow(double left, double top,
                    double right, double bottom);
void setViewport(int left, int top, int right, int bottom);
Rectangle2D getWorldWindow(void) const;
double getWorldWindowAspectRatio(void) const;
Rectangle2D getViewport(void) const;
```

The aspect-ratio of the world-window is defined as its width divided by its height. Note that when using these methods, OpenGL uses an *orthographic* projection.

2.2.2 Projection management

When moving into the realm of three-dimensional graphics, the concept of 'projection' comes into play and essentially defines how the objects in the world will be transformed to objects on the screen (that is, to objects in the viewport). A camera is defined, which can be shaped as a pyramid (for a *perspective* projection) or as a parallellipiped (for an *orthographic* projection). There is also support for an *oblique* projection, which gives a more intuitive three-dimensional appearance than a pure orthographic projection, by means of incorporating shears.

All these projections are defined in OpenGL using a so-called projection-matrix. The setup of this matrix (i.e., the setting up and controlling of the camera) can be fully automated by using the following methods :

```
void setCamera(const Point3D& cameraEyePoint,
               const Point3D& cameraViewPoint,
               const Vector3D& cameraUpDirection);
Point3D getCameraEyePoint(void) const;
Point3D getCameraViewPoint(void) const;
Vector3D getCameraUpDirection(void) const;
void slideCamera(double du, double dv, double dn);
void rollCamera(double degrees);
void pitchCamera(double degrees);
void yawCamera(double degrees);
void setCameraPerspectiveShape(double cameraViewAngleInDegrees,
                               double cameraAspectRatio,
                               double cameraDistanceOfNearPlane,
                               double cameraDistanceOfFarPlane);
void setCameraPerspectiveShape(double cameraLeftPlane,
                               double cameraTopPlane,
                               double cameraRightPlane,
                               double cameraBottomPlane,
                               double cameraDistanceOfNearPlane,
                               double cameraDistanceOfFarPlane);
void setCameraObliqueShape(const Vector3D& projectionDirection,
                           double cameraLeftPlane,
                           double cameraTopPlane,
                           double cameraRightPlane,
                           double cameraBottomPlane,
                           double cameraDistanceOfNearPlane,
                           double cameraDistanceOfFarPlane);
void setCameraOrthogonalShape(const Vector3D& projectionDirection,
                              double cameraLeftPlane,
                              double cameraTopPlane,
                              double cameraRightPlane,
                              double cameraBottomPlane,
                              double cameraDistanceOfNearPlane,
                              double cameraDistanceOfFarPlane);
double getCameraViewAngle(void) const;
double getCameraAspectRatio(void) const;
double getCameraLeftPlane(void) const;
double getCameraTopPlane(void) const;
double getCameraRightPlane(void) const;
double getCameraBottomPlane(void) const;
double getCameraDistanceOfNearPlane(void) const;
double getCameraDistanceOfFarPlane(void) const;
```

Note that for an oblique projection, a `projectionDirection` must be supplied. If this direction is $(0, 0, 1)$, then the oblique projection reverts to a classical orthographic projection. Also

note that, whilst in the world a *right-hand coordinate system* is used, the camera uses a *left-hand coordinate system*. This has the effect of looking down the negative Z-axis of the world.

```
void setCameraSlideStepSize(double cameraSlideStepSize);
void setCameraRotateStepSize(double cameraRotateStepSizeInDegrees);
void setCameraViewAngleStepSize(double cameraViewAngleStepSizeInDegrees);
void setCameraAspectRatioStepSize(double cameraAspectRatioStepSize);
void setCameraAdvanceStepSize(double cameraAdvanceStepSize);
double getCameraSlideStepSize(void) const;
double getCameraRotateStepSize(void) const;
double getCameraViewAngleStepSize(void) const;
double getCameraAspectRatioStepSize(void) const;
double getCameraAdvanceStepSize(void);
void loadCameraPath(string filename);
bool advanceCameraAlongPath(void);
void enableContinuousCameraMovements(void);
void disableContinuousCameraMovements(void);
void enableCameraMomentum(void);
void disableCameraMomentum(void);
void decoupleCameraYawAndRoll(void);
void coupleCameraYawAndRoll(void);
```

It is possible to fly the camera along a predefined-path in the world. To accomplish this, supply a filename to the `loadCameraPath()`-method. To advance the camera one step along this path, use the `advanceCameraAlongPath()`-method (which returns `false` when the end of the path is reached or `true` otherwise). The layout of such a *path-file* is very straightforward : the first number defines the total number of control-points along which the camera should pass. These control-points are given, after this number, by supplying their X-, Y- and Z-coordinates (separated by white-space) in the world. The camera flies along the control-points and its trajectory is linearly interpolated between two successive control-points. The stepsize can be controlled by using the `setCameraAdvanceStepSize()`-method.

Besides these various methods that influence the camera's appearance and moving behaviour, there are two methods, `decoupleCameraYawAndRoll()` and `coupleCameraYawAndRoll()`, which give control over the coupling of yawing with rolling. When flying around and performing a yaw, the camera is rolled proportionally with the new direction (this is the default behaviour), just as it happens in a real airplane.

2.3 Color, fog, lighting and texture management

OpenGL's lighting-model is rigidly defined. To obtain the most out of it, it is necessary to understand its inner-workings (which can be explored in [SA99]). The `TOpenGLApp`-class provides some straightforward methods for controlling this lighting-model.

2.3.1 Color

Colors are specified by using their red-, green- and blue-components (which lie in the $[0, 1]$-interval) or by using a `Color`-object (as defined in ∼**/SvenGL/opengl/primitives.h**). The background-color can take an extra parameter (its *alpha*-component) which specifies it opacity (0 indicates complete transparency, 1 indicates total opacity). When using OpenGL's lighting-model, ten material-components have to be defined : the red-, green- and blue- *ambient*-components, the red-, green- and blue- *diffuse*-components, the red-, green- and blue- *specular*-components and finally a *shininess*-coefficient (refer to OpenGL's lighting-model for more complete details). Several predefined materials are provided with the `setMaterialToXXX()`-methods. Note that an (optional) *emissive*-component can be specified (that is to say, its red-, green- and blue-components) by using the `setGlow()`-method.

When using these methods, OpenGL colors objects according to the light that shines on them. In order to produce a color that is invariant under lighting, use the `setHardColor()`-method (for example, to draw objects/strings that always need to be visible, regardless of the lighting).

Note that when specifying colors using the `Color`-class, it is possible to use the following predefined colors : `kBlack`, `kBlue`, `kGreen`, `kCyan`, `kRed`, `kMagenta`, `kDarkBrown`, `kBrown`, `kLightBrown`, `kDarkGray`, `kGray`, `kLightGray`, `kWhite` and `kYellow`.

```
void clearScreen(void) const;
void setBackgroundColor(double red,
                        double green,
                        double blue,
                        double alpha) const;
void setBackgroundColor(const Color& color) const;
void setColor(const Color& color) const;
void enableSmoothShading(void) const;
void enableFlatShading(void) const;
void setMaterial(const Color& ambientColor,
                 const Color& diffuseColor,
                 const Color& specularColor,
                 double shininessCoefficient) const;
void setMaterialToBlackPlastic(void) const;
void setMaterialToBrass(void) const;
void setMaterialToBronze(void) const;
void setMaterialToChrome(void) const;
void setMaterialToCopper(void) const;
void setMaterialToGold(void) const;
void setMaterialToPewter(void) const;
void setMaterialToSilver(void) const;
void setMaterialToPolishedSilver(void) const;
void setGlow(const Color& glowColor) const;
void dontGlow(void) const;
void setHardColor(const Color& color) const;
void setGlobalAmbientLight(const Color& ambientColor) const;
void enableExplicitViewpointCalculation(void) const;
void disableExplicitViewpointCalculation(void) const;
```

Note that the `shininessCoefficient`-parameter must lie in the $[0, 128]$-interval.

OpenGL computes specular reflection using the 'halfway-vector' $h = s + v$ and assumes by default that v is constant (namely $(0, 0, 1)$, which results in faster rendering. In order to compute the true value of v, use the `enableExplicitViewpointCalculation()`-method. Use the `disableExplicitViewpointCalculation()`-method to disable this explicit computation.

2.3.2 Fog

Fog is merely the blending of objects in the distance with a uniform 'space-filling' color. Typically, the fog is made out of a dark shade of gray. Two types of fog are supported : *linear* and *exponential*. These types refer to the method used when blending the objects' colors with the fog. This blending is based on the distance of the object (or more specifically, a part of the object) to the eyepoint. The following equation is used :

$$C = f \cdot C_{\text{object}} + (1 - f) \cdot C_{\text{fog}}, \tag{2.1}$$

in which f specifies a value (between 0 and 1) that is based on the distance-measure to use, C_{object} specifies the object's part's color and C_{fog} specifies the color of the fog.

```
void enableLinearFog(const Color& fogColor,
                     double start, double end) const;
void enableExponentialFog(const Color& fogColor,
                          double density) const;
void disableFog(void) const;
```

If *linear fog* is enabled, the following equation is used for calculating the fraction f :

$$f = \frac{e - z}{e - s} \quad (\text{with } s \neq e), \tag{2.2}$$

in which s and e specify two planes on the Z-axis that determine the fog-start en fog-end. If however, *exponential fog* is enabled, the equation for calculating the fraction f becomes :

$$f = \mathrm{e}^{-(d \cdot z)^2} \quad (\text{with } d \geq 0), \tag{2.3}$$

in which d now specifies the fog's density.

In both equations, z is the eye-coordinate distance from the eye to the object's part.

2.3.3 Lighting

Lighting in OpenGL must be explicitly enabled (i.e., switched on) in order to properly render objects according to their specified materials. It is also necessary that a number of lights is activated, which is done using the `switchLightOn()`-method. The different possible lights are : `kLight1`, `kLight2`, `kLight3`, `kLight4`, `kLight5`, `kLight6`, `kLight7` and `kLight8`.

```
void switchLightingOn(void) const;
void switchLightingOff(void) const;
void switchLightOn(int light) const;
void switchLightOff(int light) const;
void setLightPosition(int light, const Point3D& position) const;
void setLightDirection(int light, const Vector3D& direction) const;
void setLightProperties(int light,
                        const Color& ambientColor,
                        const Color& diffuseColor,
                        const Color& specularColor) const;
void makeSpotLight(int light,
                   const Vector3D& directionOfLightCone,
                   double cutOffAngle,
                   double exponent) const;
void enableLightAttenuation(int light,
                            double constantAttenuation,
                            double linearAttenuation,
                            double quadraticAttenuation) const;
void disableLightAttenuation(int light) const;
```

Each light can have specific 'material' characteristics which are used when modulating the specific components of an object's color/material.

There are three types of lights : *positional* lights, *directional* lights and (positional) *spotlights*. The first two are uniformly radiating point light-sources and they can be specified using the `setLightPosition()`- and `setLightDirection()`-methods. Spotlights only radiate inside a specified cone of light (controlled by the `directionOfLightCone`- and `cutOffAngle`-parameters). The light-strength inside this cone can also be attenuated according to a $\cos^n$-term where the n actually is the `exponent`-parameter.

For every light, it is possible to attenuate (diminish) its strength with the distance, according to the following equation :

$$\text{attenuation} = \frac{1}{k_c + k_l D + k_q D^2}, \tag{2.4}$$

in which D is the distance of the object to the light-source. The k_c, k_l and k_q are specified by the `constantAttenuation`-, `linearAttenuation`- and `quadraticAttenuation`-parameters.

2.3.4 Texture management

Applying textures to objects is relatively easy, correctly setting things up however requires some more detail. An important feature of the `TOpenGLApp`-class is the `getUniqueTextureID()`-method, which returns a unique number that OpenGL can use to identify a specific texture.

```
int getUniqueTextureID(void) const;
void enableGoodTextureInterpolation(void);
void disableGoodTextureInterpolation(void);
void enableGlowingTextures(void);
void disableGlowingTextures(void);
```

Currently, only texturing of quadrilaterals is supported (see section 2.4.3). When OpenGL is computing a texture-lookup, it can do this very straightforward (and fast) by selecting the texel whose coordinates are nearest to the center of the pixel under consideration. This may however lead to aliasing effects, which can be corrected by using a linear interpolation (with the `enableGoodTextureInterpolation()`-method), which has the disadvantage that the rendering speed will slow down.

Two modes for texturing are allowed : *glowing* and *non-glowing* textures. The first method is also known as *decal*-mode, in which the texture is just painted on the surface and no light-contributions are taken into account. The other mode, called *modulate*-mode, considers the effects the different specified lights can have. Switching between the two modes is done using the `enableGlowingTextures()`- and `disableGlowingTextures()`-methods.

2.4 Graphics routines

The graphics routines come in a wide variety, suited for two- or three-dimensional drawing. Scene-rendering is also supported (by using OpenGL's Z-buffer algorithm or `TOpenGLApp`'s ray tracing engine.

2.4.1 2D-graphics routines

A number of methods are provided for drawing points (of a specified size), lines, regular polygons and circles (which are regular polygons with a large number of edges).

```
void point(double x, double y) const;
void point(const Point2D& p) const;
void setPointSize(double pointSize) const;
void line(double x1, double y1, double x2, double y2) const;
void line(const Point2D& p, const Point2D& q) const;
void lineTo(double x, double y);
```

```
void lineTo(const Point2D& p);
void moveTo(double x, double y);
void moveTo(const Point2D& p);
void lineRel(double dx, double dy);
void moveRel(double dx, double dy);
void forward(double distance, bool isVisible);
void turnTo(double degrees);
void turn(double degrees);
void ngon(int n, double x, double y, double radius,
          double startAngleInDegrees);
void ngon(int n, const Point2D& center, double radius,
          double startAngleInDegrees);
void circle(double x, double y, double radius);
void circle(const Point2D& center, double radius);
void polyLineFromFile(string filename) const;
void drawString(int x, int y, string str, const Color& color);
```

Note that the concept of *turtle-graphics* is provided through the `forward()`-, `turn()`- and `turnTo()`-methods. An interesting idea is the `polyLineFromFile()`-method, which can read in a file (that contains a description of successive two-dimensional line-segments) and draw it. The file-format is as follows : the first number specifies the total number of polylines in the file. After this number, all the polylines are in turn successively stored as follows : first a number is specified that indicates the total number of points in the polyline. After this number all the two-dimensional X- and Y-coordinates of these points are specified. Examples can be found in the ∼**/SvenGL/data/meshes/2d**-directory.

Drawing flat text on an absolute position in the application's window, is supported by the `drawString()`-method. Note that the current color- and material-definitions are overwritten by this method (because it uses `setHardColor()`, which can be found in section 2.3.1).

2.4.2 3D-graphics routines

The three-dimensional counterparts of the routines in the previous section are roughly the same, only they operate on `Point3D`-objects (or X-, Y- and Z-coordinates).

```
void point(double x, double y, double z) const;
void point(const Point3D& p) const;
void line(double x1, double y1, double z1,
          double x2, double y2, double z2) const;
void line(const Point3D& p, const Point3D& q) const;
void lineTo(double x, double y, double z);
void lineTo(const Point3D& p);
void moveTo(double x, double y, double z);
void moveTo(const Point3D& p);
void lineRel(double dx, double dy, double dz);
void moveRel(double dx, double dy, double dz);
void cone() const;
```

```
void cube() const;
void cylinder() const;
void dodecahedron() const;
void icosahedron() const;
void octahedron() const;
void sphere() const;
void taperedCylinder(double topRadius) const;
void teapot() const;
void tetrahedron() const;
void torus(double tubeRadius, double torusRadius) const;
```

Note that three-dimensional turtle-graphics and text are not supported for drawing. Refer to section 3.3 for specific details on the scales and orientations of the cone, cube, (tapered) cylinder, dodecahedron, icosahedron, octahedron, sphere, teapot, tetrahedron and torus.

2.4.3 Miscellaneous 3D-graphics routines

Texture management was already discussed in section 2.3.4. The actual texture-mapping however, is done using the `textureQuad()`-method. Several parameters are necessary, describing which texel-coordinate to associate with each of the four three-dimensional points of the quadrilateral. An `RGBApixmap*` must also be specified, and its size must either be 96x96, 128x128 or 256x256 pixels. Note that more detailed textures require more memory and slow down the rendering process.

```
void textureQuad(const RGBApixmap* const pixmap,
                 const Point2D& tUpperLeft,
                 const Point2D& tLowerLeft,
                 const Point2D& tLowerRight,
                 const Point2D& tUpperRight,
                 const Point3D& pUpperLeft,
                 const Point3D& pLowerLeft,
                 const Point3D& pLowerRight,
                 const Point3D& pUpperRight) const;
void drawLandscape(const TerrainMap& terrainMap,
                   bool useLandscapeMaterials,
                   bool smoothenVertexNormals,
                   bool applyTextureMapping) const;
void showAxes(double length, double thickness) const;
```

Drawing landscapes is done using the `drawLandscape()`-method, which is based on the information stored in a `TerrainMap`-object (see the file **∼/SvenGL/opengl/terrainmap.h** for more details about its interface). If landscape-materials are being used (toggle this with the `kUseLandscapeMaterials`- and `kDontUseLandscapeMaterials`-constants), each polygon in the landscape is modeled with a global material, making correct OpenGL light-calculations possible. If the vertex-normals must be smoothed (toggle this with the `kSmoothenVertexNormals`- and `kDontSmoothenVertexNormals`-constants), the normals of a vertex's

six surrounding triangles are averaged, which results in super-smooth color-transitions between the landscape's polygons. If texture-mapping needs to be applied (toggle this with the `kApplyTextureMapping`- and `kDontApplyTextureMapping`-constants), the three different types of textures (sea, mountain/land en snow) of the pixmap are used. Note that when using textures, only quadrilaterals are supported, so every two triangles in the landscape are combined, resulting in a slightly 'checkered' landscape. Refer to **∼/SvenGL/landscape.cpp** to develop a more grasping intuition for working with landscapes).

A final miscellaneous method is provided for drawing the world's axes (the `showAxes()`-method). These axes are shown as small cylinders, ended with cones. The X-axis is colored red, the Y-axis is colored green and the Z-axis is colored blue.

2.4.4 Scene-rendering (OpenGL and ray tracing)

Scene-rendering is based on the SDL-language (described in chapter 3). The rendering can be done using OpenGL's own internal Z-buffering algorithm, or using the high-end ray tracing engine that the `TOpenGLApp`-class provides.

```
void loadScene(string filename);
void clearScene(void);
void setSceneViewSettings(void);
void enableSceneLighting(void) const;
void drawScene(void);
void rayTraceScene(int blockSize,
                   int nrOfConeSamples,
                   EShadowComputationMode shadowComputationMode,
                   double shadowFeelerJitterStrength,
                   EAntiAliasMode antiAliasMode,
                   bool useStochasticSampling,
                   double jitterStrength,
                   bool normalizeColors,
                   double normalizeBrightness,
                   bool findNormals,
                   double findNormalsProbability,
                   bool followRays,
                   double followRaysProbability);
long getNrOfRaysShot(void);
```

Historically, the `Scene`-class and the `TOpenGLApp`-class were constructed separately and an interface between the two had to be set up. This resulted in some methods that give both classes the ability to access each other's initialization- and result-code. Loading a scene for example, falls under initialization-code and is done using the `loadScene()`-method (the same holds for clearing a scene). To convey the new camera-settings (who can also be specified in SDL) and lighting to OpenGL, the `setSceneViewSettings()`- and `enableSceneLighting()`-methods are provided.

Drawing a scene can quickly be done by using the `drawScene()`-method, which performs OpenGL's Z-buffering.

If however, higher quality is required, this can be done by ray tracing a scene. The downside is the massive amounts of time and calculations that are spent rendering this scene, as opposed to the very quick rendering-scheme used by OpenGL. The ray tracing engine built inside the `TOpenGLApp`-class is fully explained in chapter 4. A rather large amount of parameters is provided to fine-tune the rendering of a scene. One of these parameters is the `shadowComputationMode`, which can be either `kNoShadows`, `kHardShadows` or `kSoftShadows`. Anti-aliasing can also be done (this is however fundamentally different from OpenGL's averaging), it is partially controlled by using the `antiAliasMode`-parameter, which can be `kUse1Sample`, `kUse5Samples` or `kUse9Samples`. Note however, that enabling shadow computation and anti-aliasing *significantly* slow down the rendering process.

It is possible to visualize the normals on the surfaces of the objects in a scene by setting the `findNormals`-parameter to `true` and specifying a percentage of the normals that must be shown (using the `findNormalsProbability`-parameter). In order to visualize the *secondary rays* themselves (these are the rays spawned after the primary (eye) rays), set the `followRays`-parameter to `true` and specify a percentage of the rays that must be shown (using the `followRaysProbability`-parameter).

2.4.5 2D- and 3D-transformations

Three kinds of affine transformations are supported (for both two- and three-dimensional objects). These transformations include *rotations*, *translations* and *scalings*.

```
void initCT(void) const;
void rotate2D(double degrees) const;
void rotate3D(double degrees, double x, double y, double z) const;
void translate2D(double dx, double dy) const;
void translate2D(const Point2D& p) const;
void translate3D(double dx, double dy, double dz) const;
void translate3D(const Point3D& p) const;
void scale2D(double sx, double sy) const;
void scale3D(double sx, double sy, double sz) const;
void pushCT(void) const;
void popCT(void) const;
```

OpenGL's rendering-scheme uses a so-called *model-view-matrix*, which can be modified by using the above methods. It is also possible to use a stack of these matrices, so transformations can be nested in a way. These methods are called `initCT()`, which initializes the *current transformation* (CT or model-view-matrix) with the identity matrix, `pushCT()` and `popCT()` which both control the CT's stack-behaviour.

Chapter 3

SDL

SDL is an acronym which stands for *Scene Description Language*. It is used as a file-format for conveniently storing scenes composed of three-dimensional objects. The basic SDL-syntax, as defined in [Hil01], is however extended to allow more freedom and a better usability. In this chapter, the new SDL-syntax is explained.

3.1 General layout

SDL is parsed from top-to-bottom, constructing objects (shapes) in a scene by maintaining the current-transformation (the *CT* from section 2.4.5) which is applied to every shape. This CT can be changed (much in the same way as described in that section) by using affine transformations which modify it to reflect their results.

The following two sections deal with the different SDL-keywords. In section 3.2 they are bundled together in categories for easier reference, in section 3.3 the supported shapes are considered. Note that SDL-keywords are *case-insensitive* and their parameters need not all reside on one line.

3.2 The different categories

3.2.1 Comments

A comment starts with an *exclamation-mark* (!) and continues until the end of the line. Comments can be placed after keywords as well.

3.2.2 Lights

Only two types are supported : positional point-lights and positional spotlights (thus *no* directional lights).

```
light     <px> <py> <pz> <red> <green> <blue>
spotlight <px> <py> <pz> <red> <green> <blue>
          <dx> <dy> <dz> <cut-off angle> <exponent>
```

The positions are specified using the `px`-, `py`- and `pz`-coordinates. The `red`-, `green`- and `blue`-components refer to a light's *diffuse*- and *specular*-components (no *ambient*-component is used). Refer to section 2.3.3 for more details on the parameters for spotlights (note that the `cut-off angle` is specified in degrees).

3.2.3 Transformations

Managing the current transformation (CT), in order to adjust the positions, orientations and scales of the generic shapes, can be done using the following SDL-keywords :

```
identityAffine
rotate         <angle> <ux> <uy> <uz>
translate      <dx> <dy> <dz>
scale          <sx> <sy> <sz>
transform      <x1> <y1> <z1> <x2> <y2> <z2> <x3> <y3> <z3>
push
pop
```

Their working is the same as explained in section 2.4.5, only here they are meant for three-dimensional transformations only. Note that an extra keyword is provided : `transform`. Using this keyword, it is possible to specify a more general transformation (such as for example a *shear*).

3.2.4 Boolean-objects

Boolean-objects are used in *constructive solid geometry* (CSG). They allow a rich set of combinations, based on simple shapes. The downside however is that they cannot be rendered using OpenGL's Z-buffer algorithm. They can be ray traced, but at a dramatic decrease in rendering speed. Three sorts of combinations are allowed :

```
difference   <left-child> <right-child>
intersection <left-child> <right-child>
union        <left-child> <right-child>
```

Note that the children of each of these boolean-objects can contain other materials, affine transformations, . . ., so they are typically enclosed in a (`push`,`pop`)-pair.

3.2.5 Material-properties

A full set of material-properties can be specified for each shape :

```
defaultmaterials
surfaceRoughness         <value>
emissive                 <red> <green> <blue>
ambient                  <red> <green> <blue>
diffuse                  <red> <green> <blue>
specular                 <red> <green> <blue>
specularExponent         <value>
lightBackFaces           <value>
reflectivity             <value>
glossStrength            <value>
transparency             <value>
translucencyStrength     <value>
priority                 <value>
speedOfLight             <value>
retainAmbientColor       <value>
retainDiffuseColor       <value>
retainSpecularReflection <value>
disableRefraction        <value>
```

Perturbing a surface's normal is done using the `surfaceRoughness`-keyword. The value given typically lies in $[0, 1]$. Coloring is controlled using the `emissive`-, `ambient`-, `diffuse`-, `specular`- and `specularExponent`-components, as described in section 2.3.1 (the `specularExponent`-component is the same as the `shininessCoefficient`-parameter).

When using boolean-objects, the new, visible 'inside' might not always be lit. To accomodate this, set the value following `lightBackFaces` to 1 (use 0 to disable it, as is the default).

The values for `reflectivity` and `transparency` lie in the interval $[0, 1]$. `Gloss` and `translucency` are based on jittering rays in a cone and averaging their computed colors. Refer to section 4.1.4 for more details. The `speedOfLight` lies in the interval $[0, 1]$. The priority of an object (used when transparent objects interpenetrate each other) is expressed with an integral number (higher numbers denote a higher priority), the priority of air is 0. Specify a value of 1 after the `retainAmbientColor`-, `retainDiffuseColor`-, `retainSpecularReflection`- and `disableRefraction`-keywords to enable them, specify 0 (the default) to disable them (see chapter 4 for more information on the use of these keywords).

Note that lighting of backfaces, reflectivity (and gloss) and transparency (and translucency) are only supported when ray tracing.

3.2.6 Motion blur

Motion blurring of objects in a scene can be done with `TOpenGLApp`'s ray tracing engine : the scene is rendered several times, each time applying each shape's specific motion blur transformation. Objects with motion blur disabled will stay the same, whereas objects with motion blur enabled will be *smeared out* in the final image.

Specifying a shape's motion blur behaviour (i.e., its motion blur transformation matrix) is done a bit differently from specifying its affine transformation (which will specify how the shape will look in the end) : three keywords (`mbRotate`, `mbTranslate` and `mbScale`) are supported that denote a shape's behaviour over time.

```
mbClear
mbRotate    <angle> <ux> <uy> <uz>
mbTranslate <dx> <dy> <dz>
mbScale     <sx> <sy> <sz>
mbEnable
```

Make sure to use the `mbEnable`-keyword in order to explicitly enable motion blur for the current shape. The final affine transformation will contain first the scaling, next the rotation and third the translation (that is to say, the individual transformations will be applied in that specific order). Note that after a shape is specified, the current motion blur transformation is cleared.

3.2.7 Global scene attributes

An SDL-file can contain certain information that controls the view-settings of the camera :

```
eyePoint    <x> <y> <z>
viewPoint   <x> <y> <z>
viewAngle   <value>
upDirection <dx> <dy> <dz>
```

It is also possible to specify global lighting information, such as the background color and the global ambient color of the scene (which modulates each shape's own ambient red-, green- and blue-components) :

```
background             <red> <green> <blue>
globalAmbient          <red> <green> <blue>
lightAttenuation       <constant-factor> <linear-factor>
                       <quadratic-factor>
atmosphericAttenuation <front-plane> <front-scale> <back-plane>
                       <back-scale> <red> <green> <blue>
fog                    <fog-start> <fog-end> <red> <green> <blue>
```

Light-attenuation is done as described in section 2.3.3, fog is done as described in section 2.3.1 (note that only the *linear* equation 2.2 is used). Refer to section 4.1 for more details on atmospheric attenuation (which is only supported when ray tracing).

The behaviour of `TOpenGLApp`'s ray tracing engine can be fine-tuned by using specific parameters :

```
shadowFeelerEpsilon      <value>
reflectivityEpsilon      <value>
transparencyEpsilon      <value>
meshEpsilon              <value>
adaptedShadowComputation <value>
minReflectivity          <value>
minTransparency          <value>
maxRecursionDepth        <value>
showExtents              <value>
motionBlur               <frame-length> <#time-samples>
```

All the epsilons are 10^{-6} by default. If necessary, adjust `meshEpsilon` so shading is properly computed for difficult meshes. If shadows don't seem correct, use a non-zero value for `adaptedShadowComputation`.

Shapes are perceived as reflective and transparent enough, when their respective coefficients are higher than the `minReflectivity` and `minTransparency`. To create more detailed scenes, increase the value of `maxRecursionDepth`, which is 5 by default (note that an increase can have a dramatic effect on the overall rendering speed).

To disable the showing of extents (which appear as white bars or spheres), use 0 for the value, any other number enables them.

When performing motion blur, the `frame-length` specifies over how long a time period (e.g., 1) the objects' specific motion blur transformations should be distributed. The number of `time-samples` taken for each block of pixels is by default 8. These time-samples are evenly distributed and slightly jittered over the frame-length. Each shape's motion blur transformation is completed over 1 frame.

3.2.8 Textures

Three types of texture (*solid texture*, *image texture* and *procedural texture*) are supported when ray tracing :

```
solidTexture      <type> <parameters>
imageTexture      <filename> <method> <tile-s> <tile-t>
proceduralTexture <type> <method> <parameters>
flatTextureOffset <s-offset> <t-offset>
```

The specified filename must denote an uncompressed 24-bit BMP-file. If `method` is 0 then the texture replaces the object's ambient color (no diffuse and specular components are considered), else it modulates the object's ambient and diffuse colors (this is just like *decal*- and *modulate*-mode, as described in section 2.3.4). *Note that solid textures are always modulating*.

When using non-solid-textures, the `tileS`- and `tileT`-parameters specify how many times the textures should repeat in the S- and T-directions (which are located in texture-space). The `flatTextureOffset`-keyword's parameters are two values, each between 0 and 1 (they can be used to adjust the texture in the S- and T-directions).

Note that an object can either have an image-texture or a procedural-texture, but not both. Combinations with solid-textures however are allowed. Refer to section 4.3 for more details.

Solid texture types

The following solid (three-dimensional) textures can be specified (along with their respective parameters) :

0 (none) : this is the default.

1 (checkerboard) : 3 parameters needed : the bars' scales in the X-, Y- and Z-directions.

2 (stacked colorbars) : 3 parameters needed : the bars' scales in the X-, Y- and Z-directions.

3 (RGB cubes) : 3 parameters needed : the bars' scales in the X-, Y- and Z-directions.

4 (contour-lines based on one coordinate) : 2 parameters needed : the coordinate involved (0 = X,1 = Y, 2 = Z) and the scale of the coordinate involved.

5 (contour-lines based on two coordinates) : 4 parameters needed : the coordinates involved (0 = X and Y, 1 = Y and Z, 2 = X and Z), the type of distance-measure to use (0 = Euclid, 1 = Manhattan, 2 = sin · sin, 3 = sin · cos) and the scales of the coordinates involved.

6 (contour-lines based on three coordinates) : 4 parameters needed : the type of distance-measure to use (0 = Euclid, 1 = Manhattan) and the scales in the X-, Y- and Z-directions.

7 (wood grain) : 11 parameters needed : the rings' thickness, the wobble-strength, the number of wobbles, the twist-strength, the red-, green- and blue- components of the first type of rings, the red-, green- and blue-components of the second type of rings and a flag to set smooth interpolation between different rings (if this flag is 0 only two colors are used, else the rings' colors are smoothly interpolated).

8 (marble) : 5 parameters needed : the coordinate involved (0 = X, 1 = Y, 2 = Z), the turbulence-strength (i.e. 5.0) and the weights for the red-, green- and blue-components of the marble.

9 (cow-spots) : 1 parameter needed : the black/white treshold between 0.0 and 1.0 (i.e. 0.5).

Procedural (flat) texture types

The following procedural (two-dimensional) textures can be specified (along with their respective parameters) :

0 (none) : this is the default.

1 (checkerboard) : 2 parameters needed : the number of tiles in the S- and T-directions.

2 (radial colors) : 6 parameters needed : the red-, green- and blue-components of the inside color and the red-, green- and blue-components of the outside color.

3.2.9 Macros

Macros are very convenient when using definitions that appear more then once (for example, in ∼**/SvenGL/data/sdl/include/colors.sdl** appear useful SDL-macros for several colors).

```
def <name> { <definition> }
use <name>
```

The `definition`-part will typically be enclosed in a (`push`,`pop`)-pair. Note that `name` is case-insensitive and must be defined uniquely.

3.2.10 Including other SDL-files

An easy-to-use method for separating SDL-definitions, is by putting them (logically) in separated SDL-files. These SDL-files can then be included inside other SDL-files :

```
include <filename>
```

After including the specified file, all it's definitions, lights and shapes are available to the current SDL-file (no other attributes are inherited)

3.3 The different shapes

Several *generic* shapes can be specified, some of them with extra parameters. One family of shapes (the *algebraic surfaces*) is divided into three categories : the *quadric surfaces* (which are based on second degree polynomials and provide a three-dimensional analogy of the two-dimensional conics), the *cubic surfaces* (which are based on third degree polynomials) and the *quartic surfaces* (which are based on fourth degree polynomials).

Note that texture mapping (solid texturing and flat texturing) is only supported when ray tracing.

3.3.1 Polyhedra

All polyhedra listed below, consist of convex polygons and are convex themselves :

```
buckyball
diamond
dodecahedron
icosahedron
octahedron
pyramid
tetrahedron
```

Note that internally, these shapes are actually *untriangulated* meshes.

3.3.2 Tapered cylinders

A tapered cylinder is a cylindrical body, stretching from z = 0 to z = 1, centered around the origin. It has a radius of 1 at z = 0 and a shape-specific radius at z = 1. For the `cone`, this last radius is 0, for the `cylinder`, this last radius is 1. For the `taperedCylinder`, this last radius can be freely specified.

```
cone            <distort-texture>
cylinder        <distort-texture>
taperedCylinder <small-radius> <distort-texture>
```

Note that each of these three shapes, takes an extra `distort-texture`-parameter. If this parameter is 1, then texture painted on the caps of the cone/(tapered) cylinder is *distorted* (i.e.: mapped from a circle to a square), otherwise it's shape is *retained* with the excess material *clipped*.

3.3.3 Cube, sphere and teapot

The two most commonly used shapes are the `cube` (which extends from -1 to +1 in the X-, Y- and Z-directions) and the `sphere` (which has a radius of 1). They both take no extra parameters :

```
cube
sphere
teapot
```

Notice a special shape that is provided by the `teapot`-keyword, which is based on the classical Utah-teapot (one of the very first computer graphics models). Note thas this teapot actually is a *triangulated* mesh, consisting of 3644 vertices and 6320 faces (it is stored as a *Wavefront Object File*).

3.3.4 Meshes

Meshes are a polyvalent way for representing all kinds of interesting shapes by using polygons (or sometimes more specifically triangles). They generally consist of a large number of vertices, normals and faces, which makes ray tracing a mesh rather intensive from the computational point of view (OpenGL's rendering on the contrary, can be done very fast).

```
mesh <filename (3VN, 3DV, 3DM or OBJ)> <interpolate-normals>
```

Four mesh-types are supported, the descriptions for the first three types are given below. Only the 3VN- and 3DV-filetypes are capable of storing the vertex-normals. When using the 3DM-filetype, these vertex-normals are calculated automatically. The fourth class of supported mesh-files (OBJ) consist of the *Wavefront Object Files*. Note that these models' scales' are unitized upon loading.

If you want to use a *flat-shaded mesh*, specify a 0 for the `interpolate-normals`-parameter, any other value enables *Phong-shading* of the mesh.

Structure of 3VN-files

```
    #vertices
    #normals
    #faces

list of vertices :
    X Y Z
    ...

list of normals :
    X Y Z
    ...

list of faces :
    #vertices in face
  list of vertex-indices :
      vertex-index
      ...
  list of normal-indices :
      normal-index
      ...
    ...
```

Structure of 3DV-files

```
    #vertices
    #faces

list of vertices and normals :
    vX vY vZ
    nX nY nZ
    ...

list of faces :
    #vertices in face
  list of vertex/normal-indices :
      vertex/normal-index
      ...
    ...
```

Structure of 3DM-files

```
    #vertices
    #faces

list of vertices :
    vX vY vZ
    ...

list of faces :
    #vertices in face
  list of vertex-indices :
      vertex-index
      ...
    ...
```

3.3.5 Torus

A popular shape is the torus. It is actually a quartic surface, which means that it should be classified under the algebraic surfaces. It is however provided as a distinct shape because its world-extent, specification and texture mapping are handled differently.

```
torus <tube-radius> <torus-radius>
```

The `tube-radius`-parameter refers to the radius of the smaller circle, the `torus-radius` refers to the radius of the larger circle, around which the smaller circle revolves. Note that it is possible to create a *horn torus* when `tube-radius` = `torus-radius` and a self-intersection *spindle torus* when `tube-radius` > `torus-radius` (the standard torus is also called a *ring torus*). Cross-sections of these three tori are shown in figure 3.1.

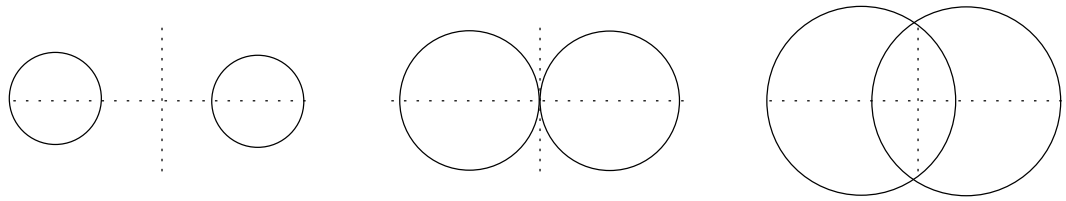

Figure 3.1: Cross-sections of the three kinds of tori.

Note that all the tori lie default in the XY-plane. It is also important to realize that ray tracing a torus is in essence solving a quartic polynomial, this means that it might be necessary to increase `shadowFeelerEpsilon` (see section 3.2.7) in order to avoid numerical instabilities.

3.3.6 Algebraic surfaces

Note that, because some of these surfaces extend infinitely, box-extents must be specified that bound them to a finite region. It is important to realize that this bounding-box is not scaled or rotated with the current affine transformation (it is only translated). An algebraic surface is thus specified as follows :

```
algebraicSurface <box-extent-width> <box-extent-height> <box-extent-depth>
```

There is *no* support for rendering the algebraic surfaces with OpenGL's Z-buffering algorithm, they can only be ray traced.

Quadric surfaces

The family of quadric surfaces supported, consists of the hyperboloid of one sheet, the hyperboloid of two sheets, the elliptic cylinder, the elliptic cone, the elliptic paraboloid and the hyperbolic paraboloid :

```
oneSheetHyperboloid  <box-extent-dimensions>
twoSheetHyperboloid  <box-extent-dimensions>
ellipticCylinder     <box-extent-dimensions>
ellipticCone         <box-extent-dimensions>
ellipticParaboloid   <box-extent-dimensions>
hyperbolicParaboloid <box-extent-dimensions>
```

Cubic surfaces

The family of cubic surfaces supported, consists of a pinched surface with holes and a pinched surface with a lobe :

```
pinchedSurfaceWithHoles <box-extent-dimensions>
pinchedSurfaceWithLobe  <box-extent-dimensions>
```

Quartic surfaces

The family of quartic surfaces supported, consists of a four-lobed shape, Steiners' Roman surface and a tooth cube :

```
fourLobes            <box-extent-dimensions>
steinersRomanSurface <box-extent-dimensions>
toothCube            <box-extent-dimensions>
```

Chapter 4

Ray tracing

The ray tracing engine which the `TOpenGLApp`-class uses, is quite powerful in that it supports various features and effects. In this chapter, the underlying lighting-model is explained in detail in order to better understand its use. Explanations of the various supported effects (of which some are based on distributed ray tracing) are elaborated. A final section is dedicated to enhancing the ray tracing engine with other shapes.

4.1 The lighting-model

The ray tracing engine casts rays from an eyepoint in directions that are specified by the distance from the near-plane to this eyepoint and the current pixel under consideration in the viewport (a perspective projection is used).

The lighting-model specifies how each ray's interactions with the objects in a scene are used. This interaction is based on the interaction of light with the objects, as described in the OpenGL lighting-model (see [SA99]). The model itself is based on three basic colors : *red*, *green* and *blue*. In the following equations, the subscript λ denotes one of these three colors.

The final color of a pixel in the application's window, will depend on a combination of several calculations. In the end, it all boils down to the following *lighting-equation* :

$$\begin{aligned} I_\lambda \;=\; & I_{\text{emissive}_\lambda} + I_{\text{ambient}_\lambda} + \\ & I_{\text{diffuse}_\lambda} + I_{\text{specular}_\lambda} + \\ & I_{\text{reflectivity}_\lambda} + I_{\text{transparency}_\lambda}, \end{aligned} \tag{4.1}$$

in which the I stands for the *intensity* of a certain component (denoted by λ).

When a ray hits an object, a lot of calculations are done to determine the final I_λ. If however, the ray doesn't hit an object, the following equation is used :

$$I_\lambda = \begin{cases} 0 & \text{if ray is reflective,} \\ c_{\text{background}_\lambda} & \text{else,} \end{cases} \tag{4.2}$$

in which $c_{\text{background}_\lambda}$ is the background color. An intensity of zero is thus returned when the ray is classified as being a secondary reflective ray. This is done in order to prevent irrealistic reflections of the background, because it is assumed that the background is not physically embodied (as opposed to, for example, the color of the air). Note that a ray is either reflective, refractive or neither and that the background is visible through transparent objects.

4.1.1 Surface roughness

The first thing that happens, is the perturbation of the surface normal according to the surface's roughness. This perturbation is based on the following random displacement :

$$\overline{\mathbf{N}'} = \overline{\mathbf{N}} + \texttt{surfaceRoughness} \cdot r_{x,y,z} - \frac{\texttt{surfaceRoughness}}{2}, \tag{4.3}$$

in which $\overline{\mathbf{N}}$ is the surface's original normal, $\overline{\mathbf{N}'}$ is the surface's perturbed normal (which is normalized after the operation) and $r_{x,y,z}$ are three random numbers (one for each coordinate-displacement) in the interval $[0, 1]$.

4.1.2 Emissive and ambient components

The emissive component is straightforward :

$$I_{\text{emissive}_\lambda} = m_{\text{emissive}_\lambda}, \tag{4.4}$$

in which $m_{\text{emissive}_\lambda}$ is the object's material's emissive component.
The ambient component is, as opposed to equation 4.4, modulated :

$$I_{\text{ambient}_\lambda} = m_{\text{ambient}_\lambda} \cdot l_{\text{global ambient}_\lambda}, \tag{4.5}$$

in which $m_{\text{ambient}_\lambda}$ is the object's material's ambient component and $l_{\text{global ambient}_\lambda}$ is the global ambient color of the scene. If the ambient color is not retained, its components are modulated according to the object's transparency (which lies in the interval $[0, 1]$) :

$$I_{\text{ambient}_\lambda} = I_{\text{ambient}_\lambda} \cdot (1 - \texttt{transparency}). \tag{4.6}$$

Note that $I_{\text{ambient}_\lambda}$ can also change depending on the texturing process, as described in section 4.3.

4.1.3 Diffuse and specular components

The diffuse component is important in that it is modulated by the light-sources in a scene, the specular component is used for generating the sharp and intens specular highlights that can be seen on surfaces :

$$I_{\text{diffuse}_\lambda} + I_{\text{specular}_\lambda} = \sum_{\forall \text{ lights i}} f_{\text{attenuation}_i} \cdot f_{\text{shadow}_i} \cdot \left(I_{\text{diffuse}_{\lambda i}} + I_{\text{specular}_{\lambda i}} \right), \quad (4.7)$$

with for each light, the diffuse and specular components calculated as follows :

$$I_{\text{diffuse}_{\lambda i}} = m_{\text{diffuse}_\lambda} \cdot l_{\text{diffuse}_{\lambda i}} \cdot (\overline{\mathbf{N'}} \bullet \overline{\mathbf{L}_i}), \quad (4.8)$$

$$I_{\text{specular}_{\lambda i}} = m_{\text{specular}_\lambda} \cdot l_{\text{specular}_{\lambda i}} \cdot (\overline{\mathbf{N'}} \bullet \overline{\mathbf{H}})^{\texttt{specularExponent}}. \quad (4.9)$$

In the above equations, $l_{\text{diffuse}_{\lambda i}}$ is the light's diffuse color component, $l_{\text{specular}_{\lambda i}}$ is the light's specular reflection component (which is actually taken to be the same as its diffuse color component), $\overline{\mathbf{N'}}$ is the surface's perturbed normal, $\overline{\mathbf{L}_i}$ is the normalized vector from the surface's intersection $\overline{\mathbf{h}}$ to the light $\overline{\mathbf{L}_{ip}}$ (i.e., $\overline{\mathbf{L}_i} = \overline{\mathbf{L}_{ip}} - \overline{\mathbf{h}}$) and is more commonly called a *shadowfeeler*, $\overline{\mathbf{H}}$ is the so-called normalized *halfway-vector*, which is $-\overline{\mathbf{V}} + \overline{\mathbf{L}_i}$ with $\overline{\mathbf{V}}$ the direction of the ray. Note that $(\overline{\mathbf{X}} \bullet \overline{\mathbf{Y}})$ denotes the vector inner-product (also called the dot product).

The factors $(\overline{\mathbf{N'}} \bullet \overline{\mathbf{L}_i})$ and $(\overline{\mathbf{N'}} \bullet \overline{\mathbf{H}})$ only give rise to relevant equations when they are strictly greater than zero. If however, lighting of backfaces is enabled, the absolute values of both factors are considered, otherwise, when negative, they are replaced by zero (so no light contribution is computed).

If the diffuse color is not retained, its components are modulated according to the object's transparency (which lies in the interval $[0, 1]$), the same holds for the specular reflection :

$$I_{\text{diffuse}_{\lambda i}} = I_{\text{diffuse}_{\lambda i}} \cdot (1 - \texttt{transparency}), \quad (4.10)$$

$$I_{\text{specular}_{\lambda i}} = I_{\text{specular}_{\lambda i}} \cdot (1 - \texttt{transparency}). \quad (4.11)$$

Note that both intensities can also change depending on the texturing process, as described in section 4.3.

In the sum of equation 4.7, the light is attenuated by the following equation :

$$f_{\text{attenuation}_i} = \min\left\{1, \frac{1}{k_c + k_l d + k_q d^2}\right\}, \tag{4.12}$$

in which k_c, k_l and k_q are the constant, linear and quadratic attenuation coefficients of the scene and d is the distance from the surface's intersection to the light under consideration.

In case the light under consideration is a spotlight, some extra work needs to be done. First of all, the angle α, between the cast shadowfeeler and the direction the spotlights shines in, is calculated. Next, a check is made to see if this angle falls within the spotlight's lightcone (described by its cutoff-angle β). If this is not the case, zero is returned as the term in the summation for this light. If however, the shadowfeeler does lie inside the lightcone, the light is attenuated according to a power of the cosine of the angle α.

The factor f_{shadow_i} in equation 4.7 is discussed in section 4.2.

4.1.4 Reflection and transparency

Reflection and transparency (refraction) are phenomena that occur when secondary rays are spawned. The spawning of these rays, depends on the `minReflectivity`-, `minTransparency`- and `maxRecursionDepth`-parameters of the scene and the `reflectivity`- and `transparency`-properties of the current object's material.

Pure reflection

The direction of the additional ray depends only on the direction of the incoming ray (the one hitting the surface). In order to prevent self-reflection, the starting point of this new ray is placed a tiny bit away from the surface, in the direction of the normal vector.

The ray's new direction $\overline{\mathbf{d}_r}$ is calculated as follows :

$$\overline{\mathbf{d}_r} = \overline{\mathbf{r}} - 2 \cdot \overline{\mathbf{N}'} \cdot (\overline{\mathbf{N}'} \bullet \overline{\mathbf{r}}), \tag{4.13}$$

in which $\overline{\mathbf{r}}$ is the direction of the incoming ray and $\overline{\mathbf{N}'}$ is the (perturbed) surface normal. The intensity $I_{\text{reflectivity}_\lambda}$ resulting from the reflection, is calculated as follows :

$$I_{\text{reflectivity}_\lambda} = I_\lambda \cdot \texttt{reflectivity}, \tag{4.14}$$

in which I_λ now stands for the intensity obtained when recursively calculating the lighting-equation for the reflected ray.

Note that reflection *inside* objects is supported.

Pure transparency

Transparency is obtained when secondary rays can pass *through* an object, penetrating it's surface. Several considerations must be taken into account when talking about transparency : are transparent objects allowed to interpenetrate, and if so, should one be completely inside of another or are they allowed to partially overlap ? The approach taken in the ray tracing engine, is that transparent objects may (partially) interpenetrate each other. This though, has the effect of specifying an extra parameter (`priority`) because inside two transparent objects, *only one medium* (which in turn defines the speed of light) is supported.

When a ray travels through a serie of objects, it contains a priority-list which tells it in which media it has traveled and thus it can be determined what the object with the highest priority is for determining the new speed of light (i.e., the *refraction index*). Note that the priority of air is 1.

The refracted ray's direction $\overline{\mathbf{d}_t}$ is calculated as follows[1] (note that, in order to prevent self-transparency, the starting point of this new ray is placed a tiny bit away from the surface, in the direction of the normal vector) :

$$\overline{\mathbf{d}_t} = \frac{c_2}{c_1} \cdot \overline{\mathbf{r}} + \left(\frac{c_2}{c_1} \cdot (\overline{\mathbf{N}'} \bullet \overline{\mathbf{r}}) - \cos(\theta) \right) \cdot \overline{\mathbf{N}'}, \tag{4.15}$$

in which $\overline{\mathbf{r}}$ is the direction of the incoming ray, $\overline{\mathbf{N}'}$ is the (perturbed) surface normal, c_1 is the speed of light in the current medium and c_2 is speed of light in the new medium (after refracting the ray). The factor $\cos(\theta)$ is calculated as follows :

$$\cos(\theta) = \sqrt{1 - \left(\frac{c_2}{c_1} \right)^2 \cdot (1 - (\overline{\mathbf{N}'} \bullet \overline{\mathbf{r}}))^2}. \tag{4.16}$$

Note that equation 4.15 is only used when the argument of the square root in equation 4.16 is greater than or equal to zero. If this is not the case, than *total internal reflection* occurs and the direction of the refracted ray becomes irrelevant (this happens at and beyond the critical angle).

The intensity $I_{\text{transparency}_\lambda}$ resulting from the refraction, is calculated as follows :

$$I_{\text{transparency}_\lambda} = I_\lambda \cdot \texttt{transparency}, \tag{4.17}$$

in which I_λ now stands for the intensity obtained when recursively calculating the lighting-equation for the refracted ray.

[1] This calculation is not used when refraction is disabled : in this case the ray travels *straight through* a body.

Gloss (blurred reflection) and translucency (blurred refraction)

The default behaviour of the ray tracing engine gives rise to sharp reflections and refractions. In real life, this is nearly an ideal case and many visible reflections and refractions appear to be blurred, 'smeared out' towards their edges. To compensate for this effect, *gloss* (blurred reflection) and *translucency* (blurred refraction) are supported. Note that using these effects *dramatically* increases the time needed to render an image.

Both techniques are implemented using '*non-weighted jittered discrete cone tracing*'. In order to fully understand this definition, it is best read backwards :

cone tracing : instead of shooting only one additional ray in the direction of reflection/refraction, a *cone of rays* is shot with the cone's central axis parallel to the original direction of reflection/refraction.

Discrete : the rays shot, are sampled on the surface of the cone and because the analytically calculated intersections are those of rays with objects and not of cones with objects, it is called *discrete* cone tracing.

Jittered : to minimize aliasing-effects, some noise is introduced by means of *jittering* the rays' directions a bit (thus by adding some random perturbations).

Non-weighted : the resulting computed intensities of all secondary rays in a cone are averaged to obtain the final intensity of the cone. All rays are given equal weights, which results in the name *non-weighted* (the non-weighted statistical mean is taken of all the secondary rays' intensities).

The number of additional rays shot, is controlled by the `nrOfConeSamples`-parameter of the `rayTraceScene()`-method. Of these rays, one is used as the cone's central ray c, the other rays r are distributed over its surface. In the left part of figure 4.1 a three-dimensional view is shown, in the middle a side-view is shown and in the right part the jittering of eight rays is clearly visible. Note that the cone's central ray c is *not* jittered.

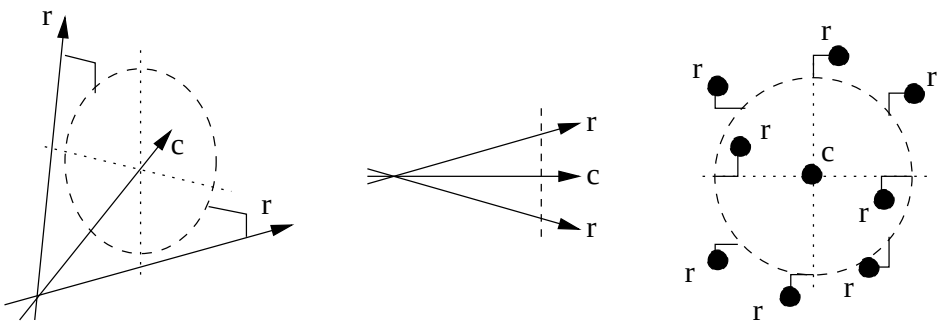

Figure 4.1: Non-weighted jittered discrete cone tracing.

The cone's radius r_c and the jittering of the rays (the jitter-strength j) are calculated as follows :

$$r_c = \frac{\texttt{glossStrength}}{2}, \tag{4.18}$$

$$j = \frac{r_c}{4}. \tag{4.19}$$

For translucency, the `glossStrength`-factor is replaced with `translucencyStrength`.

4.1.5 Atmospheric attenuation

After computing equation 4.1, atmospheric attenuation (also called *depth cueing*) is applied to the emissive, ambient, diffuse, specular, reflective and refractive components (which are commonly denoted by $I_{\text{eadsrr}_\lambda}$) :

$$I'_\lambda = s \cdot I_{\text{eadsrr}_\lambda} + (1 - s) \cdot I_{\text{attenuation}_\lambda}, \tag{4.20}$$

in which $I_{\text{attenuation}_\lambda}$ is the atmospheric attenuation color component and the scale-factor s is calculated as follows :

$$s = s_b + \frac{(d - d_b) \cdot (s_f - s_b)}{d_f - d_b} \quad (\text{with } d_f \neq d_b). \tag{4.21}$$

In the previous equation, d_f and d_b specify the front- and back-distances which correspond with certain scale-factors s_f and s_b. The Euclidean distance to the surface's intersection is denoted by $d = |\overline{\mathbf{h}} - \overline{\mathbf{E}}|$ with $\overline{\mathbf{E}}$ the eyepoint. The calculated scale s is clipped between s_f and s_b. An example of the change in s as d changes, can be seen in figure 4.2. Note that for optimal results, it is recommended that $I_{\text{attenuation}_\lambda} = c_{\text{background}_\lambda}$.

4.1.6 Fog

Fog and atmospheric attenuation can mathematically be seen as being of the same kind. However, knowing this fact, fog is explicitly built-in and it is applied to equation 4.20 :

$$I''_\lambda = f \cdot I'_\lambda + (1 - f) \cdot I_{\text{fog}_\lambda}, \tag{4.22}$$

in which I_{fog_λ} is the fog color component. The fraction f is calculated as follows :

$$f = \frac{f_e - d}{f_e - f_s} \quad (\text{with } f_s \neq f_e). \tag{4.23}$$

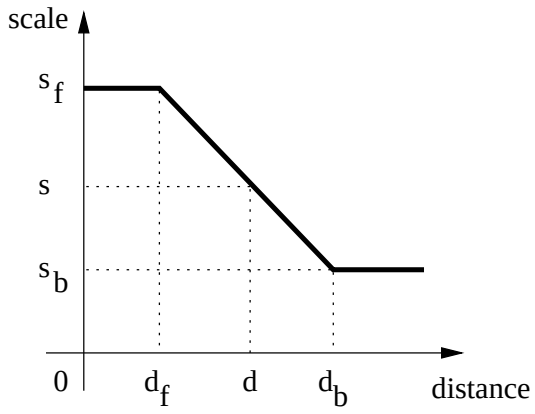

Figure 4.2: Atmospheric attenuation.

In the previous equation, f_s and f_e denote the starting- and ending-distances that specify the fog. The factor f is clipped between 0 (for distances greater than the ending-distance) and 1 (for distances smaller than the starting-distance). Note that for optimal results, it is recommended that $I_{\text{fog}_\lambda} = c_{\text{background}_\lambda}$.

The fog-calculation used here, is almost the same as that based on OpenGL in equation 2.2, but there is one big difference : OpenGL's calculations are based on the eye-coordinate distance along the Z-axis from the eye to the object's part, whilst in `TOpenGLApp`'s ray tracing engine, the real Euclidean distance is used.

4.2 Shadows

Shadows are calculated by using so-called *shadowfeelers*. They emmanate at the point where a ray intersects an object's surface and travel towards all light-sources in a scene (actually, in order to prevent self-shadowing, they are positioned a tiny bit away from the surface, in the direction of the incoming ray). If any object o lies between the start of a shadowfeeler and a certain light-source i, then the light received at the point of intersecting is tempered by the product of the transparencies t_o of all the opaque and transparent objects O in between :

$$f_{\text{shadow}_i} = \prod_{o \in O} t_o. \tag{4.24}$$

4.2.1 Hard and soft shadows

There are two kinds of shadows : *hard* shadows and *soft* shadows. The first kind is formed by spawning exactly one shadowfeeler at each intersection. The second kind is the result of spawning a cone of shadowfeelers (consisting of `nrOfConeSamples` shadowfeelers), using

non-weighted jittered discrete cone tracing (which is explained in section 4.1.4). Note that both hard and soft shadows use point light-sources (and not spheres).

Three values can be given to the `shadowComputationMode`-parameter of the `rayTraceScene()`-method : `kNoShadows`, `kHardShadows` or `kSoftShadows`.

When using soft-shadows, the cone radius r_c must be calculated. To his end, the method's `shadowFeelerJitterStrength`-parameter must be supplied. Equation 4.18 now becomes :

$$r_c = \frac{\texttt{shadowFeelerJitterStrength}}{2}, \tag{4.25}$$

$$j = \frac{r_c}{4}. \tag{4.26}$$

Note that non-convex objects (such as many meshes) can shadow themselves.

4.2.2 Indirect lighting

Indirect lighting through the use of reflective surfaces (e.g., mirrors) is not supported. For example : in reality, block B in figure 4.3 is not lit by the light-source directly but receives light through reflection of light-rays in the mirror. When using the ray tracing engine, block B resides in the shadow of block A.

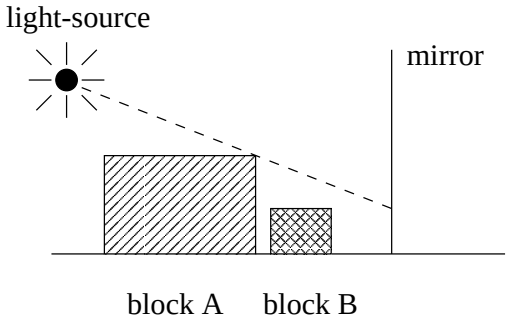

Figure 4.3: Unsupported indirect lighting.

One way to circumvent this problem, would be to perform ray tracing in the opposite direction, that is, from the light-sources towards objects in the scene (this is called *forward ray tracing* as opposed to the current method of *backward ray tracing*). This however, introduces the problem of having to shoot a very high amount of rays in order to completely cover the world-window and viewport.

4.3 Texturing

The process of texturing generally operates in two modes : replace-mode (also called *decal*-mode) and modulate-mode. Replace-mode is very little used because it doesn't take any lighting into account (i.e., glowing textures). Besides these two modes, there are also two types of textures : solid textures and flat textures. Solid textures operate on three-dimensional coordinates, whilst flat textures operate on two-dimensional coordinates. Flat textures come in two varieties : image textures and procedural textures. As mentioned in section 2.4.3, OpenGL's textures need to be of one of three predefined sizes. However, this restriction doesn't hold for the ray tracing engine, which can handle arbitrary sized images.

The general process of applying texture mapping to a point on a surface is depicted in the decision-tree, shown in figure 4.4. It can be seen that solid textures are *always modulating*. When using flat textures, replace-mode is supported, having the effect of replacing the ambient component and discarding the diffuse and specular components, which in turn results in 'always lit' textures.

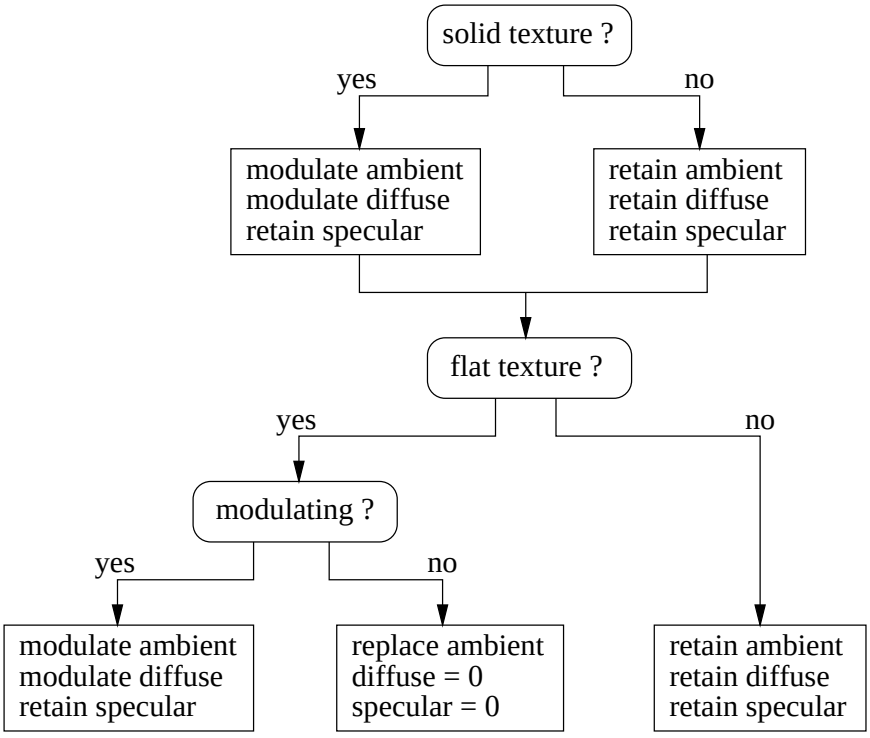

Figure 4.4: Decision-tree when applying texture mapping.

Texture-mapping is done differently for solid texturing and flat texturing, as is discussed in sections 4.3.1 and 4.3.2.

4.3.1 Solid texturing

The idea is that the texture itself resides in the world (object-space), making it possible to directly compute the texel-lookup, because its coordinates are actually those of the point under consideration. Let $t_{3D}(x, y, z)$ be the coordinate-mapping function for applying solid texture to a point (x, y, z), then :

$$t_{3D}(x, y, z) \ : \ \text{object-space}_{3D} \longrightarrow \text{texture-space}_{3D} \ : \ (x, y, z) \longmapsto (x, y, z). \tag{4.27}$$

The calculations of the coordinates are done on the generic shapes, otherwise moving objects would have shifting textures (as opposed to having them rigidly 'attached'). Because the mapping is done trivially, this has the effect that objects appear to be carved out of the solid texture.

Various textures are supported, a list of them can be found in section 3.2.8. Some of them (like the marble and cow-spots solid textures) are based on Perlin noise functions. All implementations can be found in **∼/SvenGL/opengl/solidtexturer.cpp**.

4.3.2 Flat texturing

Flat texturing is fundamentally different from solid texturing. Now, the local geometry of a shape has to be taken into account, which means that the coordinate-mapping function is (mostly) different for each shape.

In general, the following mapping is always performed when applying flat textures :

$$t_{2D}(x, y, z) \ : \ \text{object-space}_{3D} \longrightarrow [0, 1] \times [0, 1] \ : \ (x, y, z) \longmapsto (s, t). \tag{4.28}$$

After computing the (s, t) in texture-space, some housekeeping needs to be done : first, the texture-coordinate is offset (with a wrap-around). Next, the texture-coordinate is clipped in the $[0, 1]$-interval to eliminate possible rounding-errors. Finally, the texture is tiled in the S- and T-directions.

When performing texture mapping of images, additional measures need to be taken because it is possible that multiple world-coordinates map to the same texel in texture-space, resulting in a brute-force enlargement of the texture's image. To compensate this effect, bilinear interpolation of the texel-color between the four corners that surround the texel is done.

In the following paragraphs, the calculation of the texture-coordinates is explained for each shape.

Algebraic surfaces

Flat texture mapping algebraic surfaces is not easy, some of these surfaces (such as Steiner's Roman surface) aren't even orientable. In order to compensate for the plethora of algebraic surfaces and the possible mathematical difficulties, another approach to flat texture mapping them is taken : *shrink wrapping*.

Imagine a cylinder enclosing the algebraic surface's bounding-box (with b_d the bounding-box's depth) :

$$s = \frac{\mathrm{atan}(\frac{y}{x})}{2\pi}, \tag{4.29}$$

$$t = 1 - \frac{z + \frac{\mathrm{b}_d}{2}}{\mathrm{b}_d}. \tag{4.30}$$

Note how the texturing is confined to the algebraic surface's bounding-box.

Cube

Flat texture mapping a cube is straightforward : each face is mapped onto the texture-space $[0,1] \times [0,1]$. The S-direction is oriented from left to right, the T-direction is oriented from bottom to top.

Meshes

Meshes consists of a (large) number of polygons (which in particular could be triangles). Defining a unique way to flat texture map *all* meshes isn't easy. Is the texture mapping to be done on each polygon separately, or does the texture mapping needs to be done on the complete mesh ?

The approach chosen in `TOpenGLApp`'s ray tracing engine, is to perform *shrink wrapping*. It is however, done a bit differently from shrink-wrapping with algebraic surfaces. The method here is rather simple and straightforward : a point on the surface is converted from cartesian (x, y, z)-coordinates to spherical (ρ, φ, θ)-coordinates, after which the following mapping is then calculated (notice how the ρ-coordinate is discarded) :

$$s = \frac{\varphi}{2\pi}, \tag{4.31}$$

$$t = \frac{\theta}{\pi}. \tag{4.32}$$

Note that all the *polyhedra* and the *teapot* are internally processed as meshes.

Sphere

It is impossible to wrap a flat rectangle around a sphere, without distorting it in some way. The approach taken in `TOpenGLApp`'s ray tracing engine, is by converting the point on the surface from cartesian (x, y, z)-coordinates to spherical (ρ, φ, θ)-coordinates.

The following mapping is then calculated :

$$s = \frac{\varphi}{2\pi}, \tag{4.33}$$

$$t = 1 - \frac{\theta}{\pi}. \tag{4.34}$$

Tapered cylinders

Tapered cylinders mainly consist of two pieces : the wall and two caps (one of them has a radius of zero in case the tapered cylinder is a cone). Mapping the texture onto the wall is easily done :

$$s = \frac{\text{atan}(\frac{y}{x})}{2\pi}, \tag{4.35}$$

$$t = z. \tag{4.36}$$

Handling the caps is a different matter. Depending on the object's material's settings, textures are *distorted* or *clipped* when calculating the mapping from a circle to a square (in texture-space $[0, 1]\times[0, 1]$). In case the textures need to be clipped (no distortion occurs), the following mapping from a circle (with radius r) to the unit-square is used :

$$s = \frac{x + r}{2r}, \tag{4.37}$$

$$t = \frac{y + r}{2r}. \tag{4.38}$$

This means that excess parts are clipped. If all information is to be retained, the circle (with radius r) will be stretched to fit in the unit-square. Some calculations need to be done before s and t can be derived :

$$\alpha = \text{atan}(\frac{y}{x}), \tag{4.39}$$

$$r_{x,y} = \sqrt{x^2 + y^2}, \tag{4.40}$$

$$\beta = \left(\left(2 \cdot \left\lfloor \frac{\alpha}{2\pi} \right\rfloor\right) + 1\right) \cdot \frac{\pi}{4}, \tag{4.41}$$

$$\alpha' = -|\alpha - \beta| + \beta. \tag{4.42}$$

Now s' and t' can be calculated as follows :

$$0 \leq \alpha < \frac{\pi}{2} \quad \Rightarrow \quad \begin{cases} s' &= \frac{\cos(\alpha)}{\cos(\alpha')}, \\ t' &= \frac{\sin(\alpha)}{\cos(\alpha')}, \end{cases} \tag{4.43}$$

$$\frac{\pi}{2} \leq \alpha < \pi \quad \Rightarrow \quad \begin{cases} s' &= \frac{\cos(\alpha)}{\sin(\alpha')}, \\ t' &= \frac{\sin(\alpha)}{\sin(\alpha')}, \end{cases} \tag{4.44}$$

$$\pi \leq \alpha < \frac{3\pi}{2} \quad \Rightarrow \quad \begin{cases} s' &= \frac{-\cos(\alpha)}{\cos(\alpha')}, \\ t' &= \frac{-\sin(\alpha)}{\cos(\alpha')}, \end{cases} \tag{4.45}$$

$$\frac{3\pi}{2} \leq \alpha < 2\pi \quad \Rightarrow \quad \begin{cases} s' &= \frac{-\cos(\alpha)}{\sin(\alpha')}, \\ t' &= \frac{-\sin(\alpha)}{\sin(\alpha')}. \end{cases} \tag{4.46}$$

Finally, s and t can be calculated out of s' and t' :

$$s = \frac{1 + ((\frac{r_{x,y}}{r}) \cdot s')}{2}, \tag{4.47}$$

$$t = \frac{1 + ((\frac{r_{x,y}}{r}) \cdot t')}{2}. \tag{4.48}$$

Torus

Topologically, a (bounded) plane and a torus are said to be equivalent. After all, a torus can be formed by taking a rectangle, identifying two opposite ends with each other (which results in a cylinder) and bending it to identify the two circular ends.

The following mapping for s is calculated :

$$s = \frac{\text{atan}(\frac{y}{x})}{2\pi}. \tag{4.49}$$

For t, a bit more work is needed :

$$\cos(\varphi) = \frac{-\sqrt{x^2 + y^2} - r_{\text{torus}}}{r_{\text{tube}}}. \tag{4.50}$$

Based on the result obtained for $\cos(\varphi)$, the following decisions are taken :

$$\begin{aligned} \cos(\varphi) > 1 &\Rightarrow t = 0, \\ \cos(\varphi) < -1 &\Rightarrow t = \frac{1}{2}, \\ -1 \leq \cos(\varphi) \leq 1 &\Rightarrow \frac{\text{acos}(\cos(\varphi))}{2\pi} = \frac{\varphi}{2\pi}. \end{aligned} \tag{4.51}$$

Finally, if $z > 0$, then $t = 1 - t$.

4.4 Anti-aliasing

Because ray tracing is a point-sampling process (the world is sampled through a grid of pixels, defined by the window and viewport on the scene), aliasing can occur when not enough points are sampled. To compensate this, several techniques for *anti-alasing* can be used, most of which deal with *supersampling*.

The ray tracing engine that `TOpenGLApp` uses, supports four methods for performing anti-aliasing :

- regular supersampling with 5 samples/block,
- regular supersampling with 9 samples/block,
- stochastic supersampling with 5 samples/block
- and stochastic supersampling with 9 samples/block.

Note that anti-aliasing is *very time-consuming* since the total rendering time of a scene can be multiplied by a factor of five (or even nine). Most of the time, nine samples/block aren't even needed, because quite good results can be obtained when using 'only five' samples/block.

When anti-aliasing is disabled, only one ray is shot through each block's center[2]. Both regular and stochastic supersampling however, shoot additional rays (the direction of these rays depends on the technique used), see sections 4.4.1 and 4.4.2 for more details.

Controlling anti-aliasing is done using the `antiAliasMode`-parameter (which can be `kUse1Sample`, `kUse5Samples` or `kUse9Samples`), the `useStochasticSampling`-parameter (which can be `kUseStochasticSampling` or `kDontUseStochasticSampling`) and the `jitterStrength`-parameter of `TOpenGLApp`'s `rayTraceScene()`-method.

[2]Note that the ray tracing engine operates on blocks of pixels (controlled by the `blockSize`-parameter of the `rayTraceScene()`-method). If the size of a block is set to 1, the most detailed images will be obtained.

4.4.1 Regular supersampling

If the anti-alias mode is set to regular supersampling, four (or eight) nearby positions are extra sampled besides the center of the current block. The left part of figure 4.5 shows the use of five samples/block, the right part shows the use of nine samples/block. The blocks extend from block i to block $i + 1$ in the X-direction and from block j to block $j + 1$ in the Y-direction (note the top-to-bottom orientation of the Y-axis).

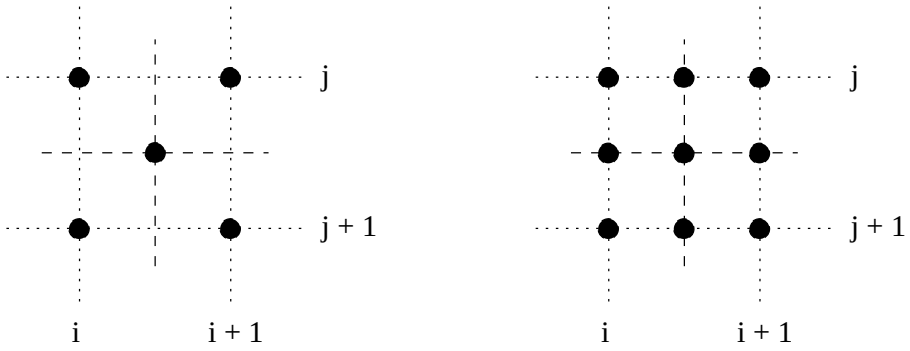

Figure 4.5: Regular supersampling.

The results, obtained by shooting the additional rays, are averaged to compute the color of the block.

4.4.2 Stochastic supersampling

Stochastic supersampling is based on regular supersampling, in that it starts from the same locations for shooting the additional rays through. However, small random perturbations are made to introduce *visual noise* (leading to *jittered* rays), which is generally better accepted by our eyes than pure regular supersampling is (which can still leave an alias of the original image visible). The left part of figure 4.6 shows the use of five samples/block, the right part shows the use of nine samples/block. The blocks extend from block i to block $i + 1$ in the X-direction and from block j to block $j + 1$ in the Y-direction (note the top-to-bottom orientation of the Y-axis).

The results, obtained by shooting the additional rays, are averaged to compute the color of the block.

The amount of random perturbation (the jitter) added to the additional ray-positions, is controlled with the `jitterStrength`-parameter of the `rayTraceScene()`-method. The method of jittering rays is based on sampling them from a minimum distance Poisson distribution, which can be approximated by jittered rays that are sampled from a jittered regular distribution.

Considering for example, the X-direction, the maximum amount of jitter, goes (for the upperleft corner of a block) from $i - \frac{e}{2}$ to $i + \frac{e}{2}$ in which e, the excess, is calculated as :

$$e = b \cdot p_w \cdot \texttt{jitterStrength}, \tag{4.52}$$

with b the block's size and p_w the width of a pixel (that is, the *real* width of a pixel on the world-window, mapped from the viewport).

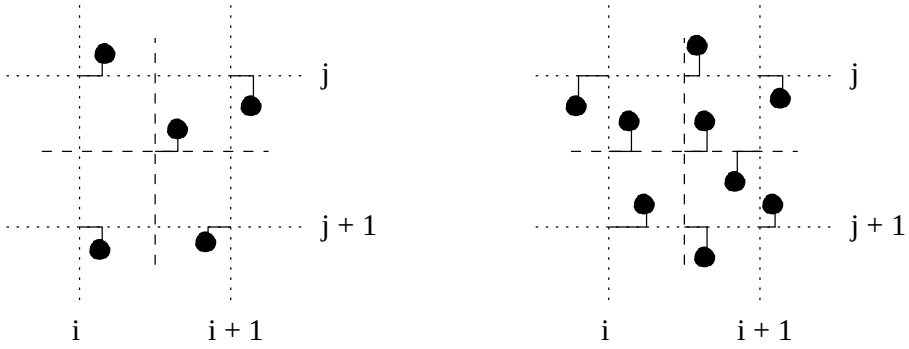

Figure 4.6: Stochastic supersampling.

4.5 Motion blur

Motion blur can be achieved when rays are temporally distributed. This distribution takes place during the period of one frame, which has a certain length (the `frame-length`-parameter accompanying the `motionBlur`-keyword from section 3.2.7). Each object also has a specific motion blur transformation, which describes the object's behaviour over time. A number of rays is evenly distributed over the frame (they are also slightly jittered).

The time-samples in a frame range from t_{start} to t_{end}, passing t_{now}, as can be seen in figure 4.7.

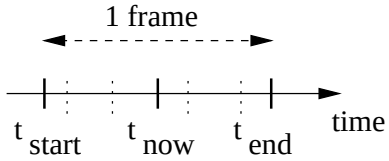

Figure 4.7: Distribution of time-samples over one frame.

The frame-edges are calculated as follows :

$$t_{\text{start}} = -\frac{\texttt{frame-length}}{2}, \tag{4.53}$$

$$t_{\text{end}} = \frac{\texttt{frame-length}}{2}. \tag{4.54}$$

When rendering with motion blur enabled, the scene is rendered several times (according to the `#time-samples`-parameter) : first, each time-sample is slightly stochastically jittered around its specific time-index, according to the `jitterStrength`-parameter of the `rayTraceScene()`-method. Next, all objects are transformed according to their own specific motion blur transformation, with respect to the current time-sample. Then the scene is rendered after which all objects are restored to their original positions, orientations and scales. Finally, after all time-samples are processed, all the rendered scenes are statistically averaged (all with the same weight).

4.6 Post-normalization of the colors

After a scene has been rendered using ray tracing, it is possible that the calculated colors of certain pixels no longer lie in the interval $[0, 1]$. In OpenGL this is simply dealt with : values higher than 1 are clipped. This has however the effect that certain portions of the rendered scene turn out to be very saturatedly colored (or even completely white). One way to deal with this problem is to keep all initial coefficients of the objects' materials' and the lights a low as possible. But this is not an option since the user then no longer has free ability to use any color he/she desires.

It is clear that this problem cannot be overcome in any other way. But it can be remedied and to that end, the ray tracing engine maintains a *render-buffer* which is an off-screen copy of the application's window. This copy can however contain 'pixels' with arbitrary intensities.

The solution to the whiteness-problem is simple : after a scene has been rendered, a search through the render-buffer is performed in order to find the highest intensity. If this highest intensity is greater than 1, all the intensities in the render-buffer are scaled so that the new highest intensity is 1. This process is called *post-normalization*. It will generally lead to darker images (when out-of-range intensities occur), and to this, end a brightness-value can be specified which increases the intensities a bit.

Post-normalization is controlled by the `normalizeColors`- and `normalizeBrightness`-parameters of the `rayTraceScene()`-method from section 2.4.4.

4.7 Adding a new shape

The need might arise to ray trace a specific, not yet supported shape. One approach is to construct the new shape using CSG (as described in section 3.2.4). This however, may not always lead to the desired result and creating a new shape from scratch could be necessary.

The following three sections clarify the steps that need to be taken when adding a new shape from scratch (for demonstration purposes, the new shape will be called `NewShape` and will have one parameter called `fParameter`, which is a `double`). After implementing the new shape, don't forget to document its usage.

4.7.1 Modifying SDL

It should be possible to use the new shape in SDL, which means that SDL needs to be extended to deal with the new shape. The following changes in several files are necessary :

~/SvenGL/opengl/scene.h : in the `private`-section of the `Scene`-class, add the constant `kNEWSHAPE` to the `ETokenType`-enumeration.

~/SvenGL/opengl/scene.cpp : two changes should be made to this file :

- in the `Scene::getGeometricObject(void)`-method, add code for handling the new shape. This `case`-code should be located inside the `switch(tokenType)`-statement at the `default`-label (where all the *shapes* are processed). For example :

```
case kNEWSHAPE :
  shape = new NewShape();
  ((NewShape*) shape)->fParameter = getDouble();
  break;
```

- In the `Scene::getToken(string keyword)`-method, add code for parsing the new shape's keyword (this code should be located in the extensive list of `if`-statements), for example :

```
else if (temp == "NEWSHAPE") { return kNEWSHAPE; }
```

  Note that the string should be completely specified in *uppercase*-characters.

4.7.2 Implementing the shape

At this point, SDL is 'aware' of the new shape, however, it still needs to be physically implemented. The general method is by constructing a new class (derived from the `Shape`-class) with some `virtual` methods overridden in order to reflect the new shape's behaviour.

Class declaration

The file ~/**SvenGL/opengl/shapes.h** should be expanded with the declaration of the new shape's class :

```
class NewShape : public Shape {
  public:
    // constructor
    NewShape(void);

    // destructor
    ~NewShape(void);

    // methods
    void drawOpenGL(void) const;
    bool computeIntersectionsWithRay(const Ray& ray,
                                     Intersection& intersection);
    void textureLookup(const HitInfo& hitInfo,
                       double& s, double& t) const;

    // data-members
    double fParameter;
};
```

Note that the `fParameter`-data-member is declared `public`.

Class definition

The file ~/**SvenGL/opengl/shapes.cpp** should be extended with the definition of the new shape's class (the implementations of all its declared methods) :

```
// --------------
// class NewShape
// --------------

// -----------
// constructor
// -----------
NewShape::NewShape(void) : fParameter(0.0)
{
}

// ----------
// destructor
// ----------
~NewShape::NewShape(void)
{
}

// -------
// methods
// -------
void NewShape::drawOpenGL(void) const
{
  tellMaterialsGL();
  glPushMatrix();
    glMultMatrixd(fAffineTransformation.m);

    // place here the OpenGL-code (or GLUT-code)
    // that renders the new shape

  glPopMatrix();
}

bool NewShape::computeIntersectionsWithRay(const Ray& ray,
                                           Intersection& intersection)
{
  Ray genericRay;
  Transformations::transformRay(genericRay,
                                fInverseAffineTransformation,ray);

  // calculate the hit-times and store them (ordered) in
  // the intersection-record
}
```

```
void NewShape::textureLookup(const HitInfo& hitInfo,
                             double& s, double& t) const
{
  Point3D p(hitInfo.fHitPoint);

  // convert the three-dimensional point p with
  // coordinates (p.fX,p.fY,p.fZ) to a two-dimensional
  // point with coordinates (s,t) with 0 <= s,t <= 1
}
```

The most difficult part usually lies in developing the intersection-code for the ray tracing engine. Refer the ∼**/SvenGL/opengl/shapes.cpp**-file where good, helpful examples can be found.

Note that the `textureLookup()`-method only needs to perform the conversion from three-dimensional coordinates to two-dimensional coordinates (offsetting, tiling and clipping are automatically done).

4.7.3 Using a world-extent

Sometimes, the use of world-extents might provide a vaste increase in rendering speed when ray tracing. As an example, rendering a torus without a world-extent can take ten times longer than rendering one with a world-extent.

Using extents is very straightforward, continuing the example from the previous sections, the following files must be altered :

∼**/SvenGL/opengl/shapes.h** : a method for computing the world-extent should be declared in the class's interface :

```
class NewShape : public Shape {
  public:
    // ...

    // methods
    void computeExtent(void);
```

~/SvenGL/opengl/shapes.cpp :

- the previously declared `computeExtent()`-method must be implemented :

```
void NewShape::computeExtent(void)
{
  // create the point-cluster for
  // the generic user-defined-shape
  PointCluster pointCluster(2);
  pointCluster.fPoints[0] = Point3D(-1.0,0.0,0.0); // e.g.
  pointCluster.fPoints[1] = Point3D(1.0,0.0,0.0); // e.g.

  // apply the NewShape's specific affine transformation
  // to express the point-cluster in world coordinates
  // instead of generic shape coordinates
  pointCluster.applyAffineTransformation(fAffineTransformation);

  // note that instead you can only perform the translation-part
  // of the specified affine transformation, which retains the
  // shape of the generic extent
  // to do this, use :
  //   pointCluster.applyTranslation(fAffineTransformation);

  // e.g.: we use the sphere-extent
  fWorldExtent = new SphereExtent();
  // use a small excess for the extent (e.g. 0.1)
  fWorldExtent->create(pointCluster,0.1);
}
```

  Computing a world-extent is done using a *point cluster*. Use enough points so the `NewShape` is completely enclosed. Then settle for either a *box-extent* or a *sphere-extent* (or create a specifically tailored extent). Use a small excess so the world-extent doesn't fit to tight around the `NewShape`.

- Finally, an extent-intersection-test should be included :

```
bool NewShape::computeIntersectionsWithRay(const Ray& ray,
                                Intersection& intersection)
{
  // perform extent-intersection-test
  // (in world-coordinates !)
  if (!fWorldExtent->rayIntersectsExtent(ray)) {
    return false;
  }

  // rest of code...
}
```

Appendix A

Utilities

In this appendix, various utilities are presented that may assist in programming a GL-class. A tool for measuring time (`Chrono`), a collection of mathematical routines (`Math`) and a method for generating random numbers (`RNG`) are provided.

In order to use these classes, they should be included in the file where they are needed, using some or all of the following statements :

```
#include "opengl/chrono.h"
#include "opengl/math.h"
#include "opengl/rng.h"
```

A.1 Chrono

The class `Chrono` provides a chrono-meter for measuring elapsed times (with an accuracy of one second). It is based on the following methods :

```
void start(void);
void stop(void);
long getElapsedTime(void);
static string convertToHMS(long elapsedTime);
```

The `static convertToHMS()`-method converts the `long`-value returned by the `getElapsedTime()`-method to a string in the *hh:mm:ss*-format (with *hh* the number of elapsed hours, *mm* the number of elapsed minutes and *ss* the number of elapsed seconds).

A.2 Math

The `Math`-class provides various frequently needed constants and operations :

A.2.1 Constants

```
static const double kPi;
static const double kEpsilon;
static const double kInfinity;
```

Using these constants is done by prefixing them with the scope-operator, for example :

```
Math::kEpsilon
```

A.2.2 Operations

The following mathematical operations are provided :

```
static double abs(double x);
static double frac(double x);
static double sqr(double x);
static double cube(double x);
static double quart(double x);
static double cubicRoot(double x);
static double lerp(double a, double b, double fraction);
static double arctan(double y, double x);
static double radToDeg(double radians);
static double degToRad(double degrees);
static void convertCartesianToSpherical(double x, double y, double z,
                                        double& radius, double& phi,
                                        double& theta);
static int solveQuadraticEquation(double coefficients[3],
                                  double solutions[2]);
static int solveCubicEquation(double coefficients[4],
                              double solutions[3]);
static int solveQuarticEquation(double coefficients[5],
                                double solutions[4]);
```

Using these operations is done by prefixing them with the scope-operator, for example :

```
Math::degToRad(45.0)
```

The following conventions hold :

$$\begin{aligned}
\texttt{abs(x)} &= |x|, && \text{(A.1)}\\
\texttt{frac(x)} &= x - \lfloor x \rfloor, && \text{(A.2)}\\
\texttt{sqr(x)} &= x^2, && \text{(A.3)}\\
\texttt{cube(x)} &= x^3, && \text{(A.4)}\\
\texttt{quart(x)} &= x^4, && \text{(A.5)}\\
\texttt{cubicRoot(x)} &= \sqrt[3]{x}, && \text{(A.6)}\\
\texttt{lerp(a,b,fraction)} &= a + (\texttt{fraction} \cdot (b - a))), && \text{(A.7)}\\
\texttt{arctan(y,x)} &= \text{atan}\left(\frac{y}{x}\right), && \text{(A.8)}\\
\texttt{radToDeg(r)} &= \frac{180\, r}{\pi}, && \text{(A.9)}\\
\texttt{degToRad(d)} &= \frac{d\, \pi}{180}. && \text{(A.10)}
\end{aligned}$$

The coefficients for the `solveXXX()`-methods, describe their polynomials in the following way :

$$c_0 + c_1 \cdot x + c_2 \cdot x^2 + c_3 \cdot x^3 + c_4 \cdot x^4. \qquad \text{(A.11)}$$

A.3 RNG

With the use of the `RNG`-class, a powerful multiply-with-carry type of random number generator (invented by George Marsaglia) is made available. The algorithm used, is :

$$\begin{aligned}
S &= (2111111111 \cdot X[n-4]) + \\
&\quad (1492 \cdot X[n-3]) + \\
&\quad (1776 \cdot X[n-2]) + \\
&\quad (5115 \cdot X[n-1]) + C, && \text{(A.12)}\\
X[n] &= S \bmod 2^{32}, && \text{(A.13)}\\
C &= \left\lfloor \frac{S}{2^{32}} \right\rfloor. && \text{(A.14)}
\end{aligned}$$

Note that the value of C needs to be stored in a 80-bits `long double`.

The following `static` methods are provided :

```
static void setSeed(uint32 seed);
static void setSeedWithTime(void);
static int uniformInt(void);
static double uniform(void);
```

Note that `uint32` is an `unsigned long`.

Calling these methods is done by prefixing them with the scope-operator, for example, to initialize the random number generator, use the following statement :

```
RNG::setSeedWithTime();
```

Appendix B

Key-assignments

In this appendix, the various keys that control the `TOpenGLApp`-class are explained. There are two sections : the first one lists the keys that are already assigned whilst the second one lists the available keys.

B.1 Assigned keys

The following keys are used by the `TOpenGLApp`-class :

left/right : yaw camera left/right,

up/down : pitch camera down/up,

PGUP/PGDN : roll camera left/right,

m/M : disable/enable camera momentum,

z/Z : decrease/increase camera's slide-stepsize,

y/Y : decouple/couple camera's yawing from/with rolling,

v/V : decrease/increase camera's view-angle,

a/A : decrease/increase camera's aspect-ratio,

n/N : decrease/increase distance of near-plane,

f/F : decrease/increase distance of far-plane,

l/L : turn lighting off/on,

s/S : enable flat/smooth shading,

e/E : disable/enable anti-aliasing,

x/X : toggle between windowed mode/full screen,

t/T : take a snapshot of the scene (save to **snapshotX.bmp** with $X \in \{0, 1, \ldots\}$),

ESCAPE : exit program.

When the camera is *not* in momentum-mode, the following keys are also available :

F1/F2 : decrease/increase camera x-position,

F3/F4 : decrease/increase camera y-position,

F5/F6 : decrease/increase camera z-position,

F7/F8 : slide camera left/right (along its own x-axis),

F9/F10 : slide camera forward/backward (along its own y-axis),

F11/F12 : slide camera forward/backward (along its own z-axis),

c/C : disable/enable continous camera movements.

B.2 Available keys

The following alphanumeric keys are available : r/R, u/U, i/I, p/P, q/Q, d/D, g/G, h/H, j/J, k/K, w/W and b/B. Note that the numeric keys (0 . . . 9) are also available.

References

[3DC01] 3DCafe. *3D Cafe's Free 3D Models Meshes*. Webextract – Platinum Pictures Multimeda, Inc., 2001.
(URL : `http://www.3dcafe.com/meshes/meshes.asp`).

[ART01] ARTLab. *OpenGL – High Performance 2D/3D Graphics*. Webextract – Silicon Graphics, Inc., 2001.
(URL : `http://www.opengl.org`).

[Arv86] James Arvo. Backward ray tracing. In *Developments in Ray Tracing*, pages 259–263. ACM Siggraph Course Notes 12, 1986.
(URL : `http://www.cs.caltech.edu/~arvo/papers.html`).

[BHH00] I. Buck, G. Humphreys, and P. Hanrahan. *Tracking graphics state for networked rendering*, 2000.
(URL : `http://graphics.stanford.edu/papers/state_tracking/`).

[Bry00] Jason Bryan. *Advanced 3D Image Generation*. Webextract – The Ohio State University, 2000.
(URL : `http://www.cis.ohio-state.edu/~jbryan/School/cis782`).

[Bus00] Sam Buss. *Ray Tracing Samples*. Webextract – University of California, San Diego, 2000.
(URL : `http://www.math.ucsd.edu/~sbuss/Math155Fall99Winter00/17_RayTracing.html`).

[CK01] Zong Da Chen and Adriana Karagiozova. *Rendering and Morphing of Quartic Surfaces*. Webextract – Harvard University, 2001.
(URL : `http://www.fas.harvard.edu/~lib175/projects_fall_2000/karagioz/project275.html`).

[CL95] Marshall P. Cline and Greg A. Lomow. *C++ FAQs – Frequently Asked Questions*. Addison-Wesley Publishing Company, One Jacob Way, Reading, Massachusetts 01867, 1995. ISBN 0-201-58958-3.

[Cla01] Roger Claw. *Imperial Archive for Textures and Surfaces*. Webextract – The Imperial Design Bureau, 2001.
(URL : `http://atlas.spaceports.com/~admiral`).

[Cof01] Adam Coffman. *Steiner Surfaces*. Webextract – Indiana University Purdue University Fort Wayne, 2001.
(URL : `http://www.ipfw.edu/math/Coffman/steinersurface.html`).

[CS] Swen Campagna and Philipp Slusallek. *Improving Bezier Clipping and Chebyshev Boxing for Ray Tracing Parametric Surfaces*. Computer Graphics Group, University of Erlangen, Germany.

[CSS] Swen Campagna, Philipp Slusallek, and Hans Peter Seidel. *Ray Tracing of Parametric Surfaces : Bezier Clipping, Chebyshev Boxing and Bounding Volume Hierarchy. A Critical Comparison with New Results*. Computer Graphics Group, University of Erlangen, Germany.

[DH92] J. Danskin and P. Hanrahan. *Fast Algorithms for Volume Ray Tracing*, 1992.
(URL : `http://graphics.stanford.edu/papers/volume/`).

[Eli00] Hugo Elias. *Perlin Noise*. Webextract, 2000.
(URL : `http://freespace.virgin.net/hugo.elias/models/m_perlin.html`).

[FSJ] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. *Visual Simulation of Smoke*. Stanford University, Gates Computer Science Building, Stanford, CA 94305-9020.
(URL : `http://graphics.stanford.edu/~henrik/papers/smoke/`).

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley Systems Programming Series. Addison-Wesley Publishing Company, One Jacob Way, Reading, Massachusetts 01867, 2nd edition, 1990. ISBN 0-201-12110-7.

[Gan01] Xiao Gang. *WIMS Interactive Mathematics Server*. Webextract – Université de Nice Sophia-Antipolis, 2001.
(URL : `http://wims.unice.fr/~wims`).

[GTS] Donald P. Greenberg, Kenneth E. Torrance, and Peter Shirley. *A Framework for Realistic Image Synthesis*. Program of Computer Graphics, Cornell University, 580 Frank H.T. Rhodes Hall, Ithaca, NY 14853.

[Hai01] Eric Haines. *Graphics Gems Repository*. Webextract – ACM (Association for Computing Machinery), 2001.
(URL : `http://www.graphicsgems.org`).

[Har01] John C. Hart. *Advanced Topics in Computer Graphics*. Webextract – University of Illinois at Urbana-Champaign, 2001.
(URL : `http://www-courses.cs.uiuc.edu/~cs319`).

[Hil01] Francis S. Hill. *Computer Graphics Using Open GL*. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, 2nd edition, 2001. ISBN 0-02-354856-8.

[Hun00] Bruce Hunt. *Gallery of Algebraic Surfaces*. Webextract – Universität Kaiserslautern, 2000.
(URL : `http://www.mathematik.uni-kl.de/~wwwagag/D/Galerie.html`).

[IRT01] IRTC. *The Internet Raytracing Competition*. Webextract – IRTC Administration Team, 2001.
(URL : `http://www.irtc.org`).

[KA88] David Kirk and James Arvo. The ray tracing kernel. In *Proceedings of Ausgraph '88*, pages 75–82, Melbourne, Australia, July 1988.

[Kaj83] James T. Kajiya. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, july 1983. California Institute of Technology.

[Kil96] Mark J. Kilgard. *The OpenGL® Utility Toolkit (GLUT) Programming Interface*. Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311, November 1996. API Version 3.

[KMH] Craig Kolb, Don Mitchell, and Pat Hanrahan. *A Realistic Camera Model for Computer Graphics*. Computer Science Department (Princeton University), Advanced Technology Division (Microsoft), Computer Science Department (Stanford University).

[LKR+96] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH '96*, pages 109–118, 1996.
(URL : `http://www.gvu.gatech.edu/people/peter.lindstrom/papers/siggraph96/`).

[MCFS00] William Martin, Elaine Cohen, Russell Fish, and Peter Shirley. Practical ray tracing of trimmed nurbs surfaces. *Journal of Graphics Tools*, 5(1):27–52, 2000.
(URL : `http://www.cs.utah.edu/vissim/papers/raynurbs/`).

[Met00] Jonathan Metzgar. *SuperQuadric Ellipsoids and Toroids, OpenGL Lighting, and Timing*. Webextract – Microwerx Systems, 2000.
(URL : `http://www.geocities.com/microwerx/projects/superquadric.html`).

[Nor97] Tore Nordstrand. *Gallery of Mathematical Surfaces*. Webextract – University of Bergen, Norway, 1997.
(URL : `http://www.uib.no/People/nfytn/mathgal.htm`).

[NYH01] T. Naemura, T. Yoshida, and H. Harashima. 3-d computer graphics based on integral photography. *Optics Express 255*, 8(2), february 2001. Dept. of Inform. & Commun. Eng., The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan.

[Per97] Ken Perlin. *Noise and Turbulence*. Webextract – New York University Media Research Lab, 1997.
(URL : `http://mrl.nyu.edu/~perlin/doc/oscar.html`).

[Pix00] Pixar. *The RenderMan® Interface*, July 2000. Version 3.2.

[PKGH97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, 1997.

[PMS+99] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 119–126, Atlanta, GA USA, April 1999.
(URL : `http://www.acm.org/pubs/citations/proceedings/graph/300523/p119-parker/`).

[PPD98] Eric Paquette, Pierre Poulin, and George Drettakis. A light hierarchy for fast rendering of scenes with many lights. In N. Göbel and F. Nunes Ferreira (guest editor), editors, *Computer Graphics Forum (Eurographics '98 Conference Proceedings)*, pages 63–74. Eurographics, September 1998. held in Lisbon, Portugal, 02-04 September 1998
(URL : `http://www-imagis.imag.fr/Publications/1998/PPD98`).

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 233–238, 1998.
(URL : `http://www.cs.utah.edu/~shirley/papers/iso/`).

[RJ97] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7):873–885, 1997.
(URL : `http://www.cs.bris.ac.uk/Tools/Reports/Abstracts/1997-reinhard.html`).

[RSH] Erik Reinhard, Brian Smits, and Charles Hansen. *Dynamic Acceleration Structures for Interactive Ray Tracing*.
(URL : `http://www.cs.utah.edu/~reinhard/egwr/`).

[SA99] Mark Segal and Kurt Akeley. *The OpenGL® Graphics System : A Specification*. Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311, April 1 1999. Version 1.2.1.

[Sch] Christophe Schlick. *An Inexpensive BRDF Model for Physically Based Rendering*. Eurographics Proceedings, Computer Graphics Forum, Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux.

[Sto95] J. Stone. *An Efficient Library for Parallel Ray Tracing and Animation*, 1995. (URL : `http://jedi.ks.uiuc.edu/~johns/raytracer/papers/isug95/`).

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, One Jacob Way, Reading, Massachusetts 01867, 3rd edition, 1997. ISBN 0-201-88954-4.

[SW92] Peter Shirley and Changyaw Wang. Distribution ray tracing : Theory and practice. In *Third Eurographics Workshop on Rendering*, pages 33–43, Bristol, UK, 1992. (URL : `http://www.cs.utah.edu/~shirley/papers/rw92/`).

[TCRS00] M. Tarini, P. Cignoni, C. Rocchini, and R. Scopigno. Real time, accurate, multi-featured rendering of bump mapped surfaces. In *Eurographics Proceedings*, volume 19-3, Istituto Elaborazione dell Informazione, Pisa, Italy, 2000. Eurographics Association and Blackwell Publishers 2000.

[The01] TheForce.net. *Scifi 3D – High Quality Meshes*. Webextract, 2001. (URL : `http://www.theforce.net/scifi3d`).

[Tig99] Mark Tigges. *Two Dimensional Texture Mapping of Implicit Surfaces*. Master's thesis, Department of Computer Science – The University of Calgary, Calgary, Alberta, June 1999.

[Tur] Greg Turk. *Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion*. University of North Carolina at Chapel Hill.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20. Blackwell Publishers, Oxford, 2001. (URL : `http://graphics.cs.uni-sb.de/~wald/Publications`).

[WS01] Ingo Wald and Philipp Slusallek. *State of the Art in Interactive Ray Tracing*. Computer Graphics Group, Saarland University, The Eurographics Association, 2001.

[WSB] Ingo Wald, Philipp Slusallek, and Carsten Benthin. *Interactive Distributed Ray Tracing of Highly Complex Models*. Computer Graphics Group, Saarland University, Saarbruecken, Germany.
(URL : `http://graphics.cs.uni-sb.de/RTRT/Gallery/`).

[Zha00] Xiaoli Zhang. *Distributed Ray Tracing*. Webextract – Simon Fraser University, 2000.
(URL : `http://www.sfu.ca/~xzhanga/cgproresult.html`).