BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

Realtime shadow algorithms

MSc. Thesis

2006 **Tomáš Burian**

Realtime shadow algorithms

Submitted to Brno University of Technology Faculty of Information Technology on 17th May 2006

. . . .

Tomáš Burian

17th May 2006

Abstract

This diploma work deals with possibilities of real-time shadow rendering in Open Inventor. This feature is still missing in this high-level 3D graphics toolkit by now. The goal was to create a library that adds shadows to user scene objects and it will automatically detect situations, when shadows must be actualised. Shadows are computed by shadow volumes method that ensures geometrically correctness and other important properties of generated shadows.

The library and all demo applications are developed in C++ language with OpenGL graphic interface and Open Inventor library.

Keywords

Shadow, shadow volume, depth-pass, depth-fail, engine, OpenGL, Open Inventor, stencil buffer.

Acknowledgements

I would like to thank to my supervisor, Ing. Jan Pečiva, for his ideas, advice and leadership and to Mr. David Ambrož and Mr. Hun Yen Kwoon for their permission to reprint their images.

Contents

Chapter 1

Preface

1.1 Introduction

Realistic shadows are important components of 3D scene. They support perception of three dimensional reality projected on 2D screen and help man to recognise positions of objects in space, as shown in the figure 1.5, their shapes, watch the figure 1.2, and dimensions. According to shadows we can also guess position of a light source and its properties.

This way shadows make scenes more realistic and thus shadow techniques take a relevant place in computer graphic.

However, there are two main reasons, why generating shadows in real time is a hard task for computers. At first, the shadow generating algorithm should be robust and create correct shadows for all possible situations. There is a couple of methods, which generate correct shadows only *under certain conditions*, e.g. planar shadows, or create incorrect shadows.

Another thing, even more important, is speed of shadow algorithms. Realtime applications such as games or virtual reality need fast algorithms which produce correct results and do not load CPU too much. Therefore these algorithms are often implemented directly in graphical processing units, vertex shaders etc.

This paper describes implementation of shadow algorithms in *Open Inventor*, or more precisely in its fully compatible GPL clone, Coin3D [1]. Until now, there was not any library or tool which could add shadows to this popular high-level 3D graphics toolkit. Thus a shadow engine with simple API was developed to fulfil this blank space.

Among many of shadow generating methods, which are briefly described in the next chapter (2), the *Shadow volumes method* has been chosen for its robustness, hardware support and possible usage in real-time.

The rest of this chapter lists different types of shadows and shadows properties.

The chapter (3) describes *Shadow volumes method* in more detail, lists pros and cons of depth-pass and depth-fail algorithms and compares them in the end.

Design of the *Shadow Engine* API, its classes and information flow is described in chapter (4).

Chapter (5) deals with implementation of *Shadow volumes method* and the *Shadow Engine* in Coin3D. How to incorporate the *Shadow Engine* into Coin3D applications is described in chapter (6). Also the usage of developed shadow demo applications can be found here.

Achieved results are shown and described in chapter (7) and plans for the future are mentioned in conclusion (8).

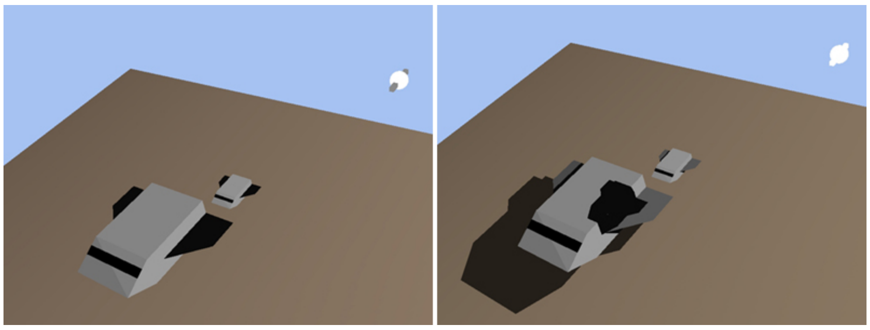
Figure 1.1: Shadows give information about reciprocal position of objects in the scene. Without shadows (left image) it is impossible to say, that the bigger rocket is lower than the small one. [Shadow Engine]

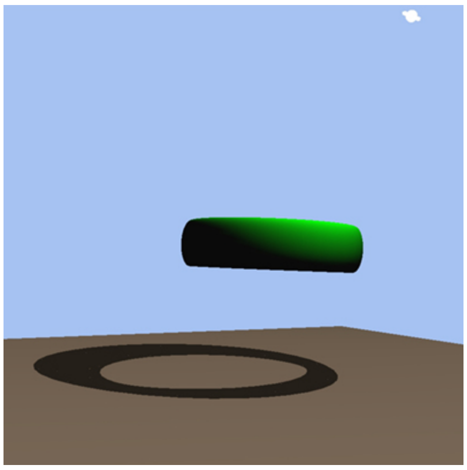
Figure 1.2: Only thanks to the shadow one can say, that the object has a hole inside. [Shadow Engine]

1.2 Shadow types

Shape and dimensions of shadows depend on reciprocal position of a light source, shadow casting object (occluder) and object the shadow is shown on (shadow receiver). Shape, size and type of a light source influence the character of a shadow as well.

Shadows can be divided according to two main aspects. The first one is, which places the shadow hides from a light, and the second is, the type of a light source, that creates shadow.

Cast and self-shadows

Shadows can be differed into two basic types: *cast shadow* and *self shadow*. The first one is a shadow of an object shown on another object or objects. This type of shadow helps understand placement of occluder and receivers in space.

The other type of shadow is a shadow placed directly on the shadow casting object itself. This is called *self-shadowing*. Not all algorithms can solve this type of shadows. The figure 1.3 shows a nice example of *self-shadowing*.

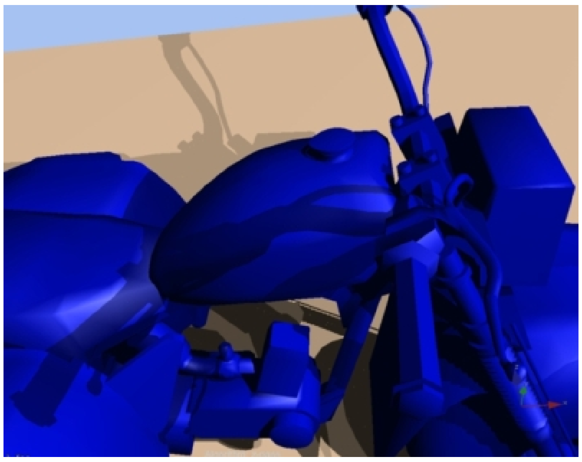

Figure 1.3: Self-shadow of the handlebar copying the surface of the motorbike. The light-grey shadow in the background is a cast shadow. [Shadow Engine]

Hard and soft shadows

There exist only area light sources in the real world, which casts *soft shadows*. For example, the sun is a typical area light source. Therefore, soft shadows created by the sun consists from two parts: *umbra* and *penumbra* as shown in the figure 1.4.

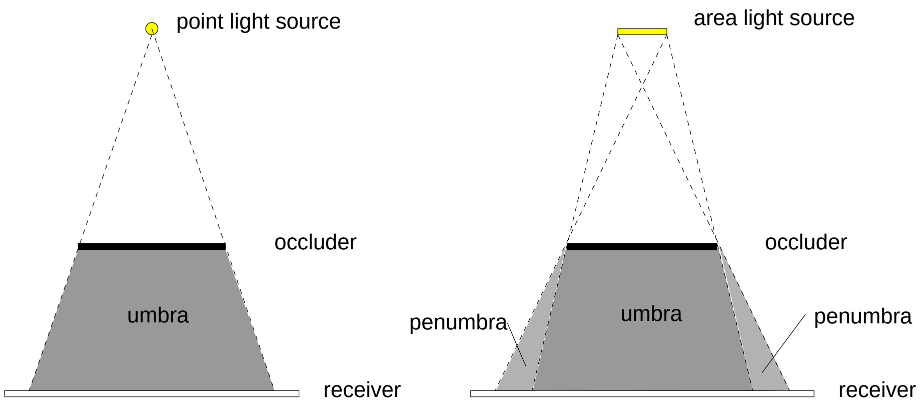

Figure 1.4: *Hard shadow* created by a point light (left image) and *soft shadow* of an area light source.

Penumbra is a gradual transition from full shadow (umbra) to lighted place. Whereas square of penumbra grows with larger area of area light source, umbra is getting smaller and can disappear completely.

However, area light sources are often replaced by point light sources in computer graphics. This simplification is done because of high computing complexity of soft shadows.

Point light sources creates *hard shadows*, which do not have smoothed edges and the border of the shadow is outlined exactly. Hard shadows look a bit unrealistic in comparison with *soft shadows* as shown in the figure 1.5.

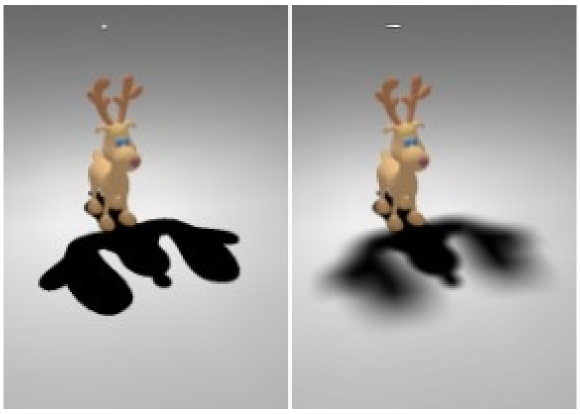

Figure 1.5: Difference between hard (left image) and soft (right image) shadows of the same object.

Obviously, there exist techniques which can create *soft shadows*, for example one area light source can be replaced by a number of point light sources to achieve a soft shadow. More on this can be found in [8] and [13].

1.3 Shadow superposition and colour

Superposition of shadows

In a general case, when a number of shadows cast onto a one surface, the resulting shadow can be computed as a unification of all shadows. There are no problems if all shadows are hard ones. In the case of superposition of soft shadows of a number of casting objects, incorrect results can be obtained. This can happen, because soft shadows do not have sharp boundaries, and the places, from which the light source cannot be seen, are not defined exactly.

Colour of shadows

Shadow is area of a scene hidden from a light source. Therefore, its colour should be black. But shadows are not completely dark, their darkness depends on the amount of ambient light[1] in the scene. So the shadowed regions should be of a colour of a scene rendered only with ambient light.

This is true if there is only one light source in the scene. When there are more than one light source (can be of different colours each), a shadow of a one object and one light can be lighted by other lights. That means, that its colour is not influenced by ambient light only, but it must be computed as a composition of ambient light and all lights that shine on the shadow area as shows the figure 1.6.

[1]Ambient light is light that is uniformly spreads through the scene. It comes from all directions. Even if no light is present in the scene, the ambient light makes the scene not black.

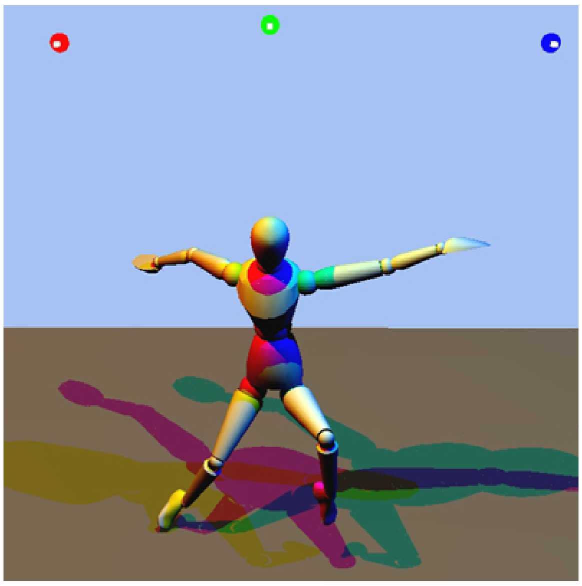

Figure 1.6: Intersections of three point light sources shadows of different colours. Note that the intersection of all three shadows is dark, of ambient light colour. [Shadow Engine]

Chapter 2

Shadow algorithms

The nowadays most used shadow algorithms in computer graphic are shortly referred in this chapter. Their properties, advantages and disadvantages are pointed out. *Shadow volumes method*, which has been chosen to be implemented in the developed *Shadow Engine*, is described in the next chapter (3) in detail.

2.1 Global illumination models

Global realistic illumination models, e.g. radiosity, computes shadows automatically. This method is based on modelling of reciprocal emission and absorbing of photons[1] in the static scene.

Its very realistic result shows the figure 2.1. The main advantages of radiosity are view-independent model of photons emission, that can be for a particular scene pre-computed. It also models transparent objects, which can be difficult for other methods.

Hardware demands of this method are, however, too high and thus radiosity is not widely used in real-time applications.

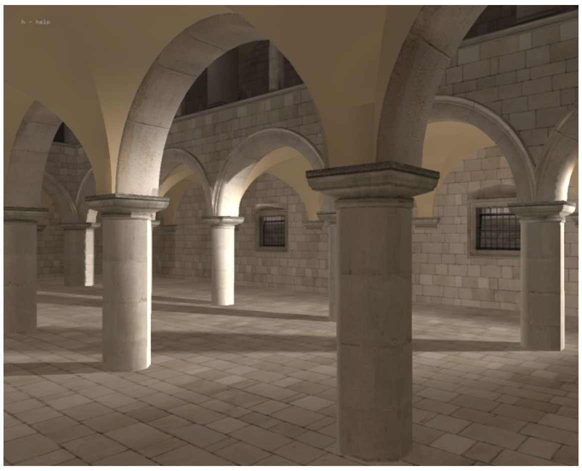

Figure 2.1: Scene created by radiosity method. Realistic look of the image takes a long computation time.

[1] A photons are light energy particles which are spread from light sources to the scene. When they hit a surface of an object, a part of their energy is absorbed and the rest is again emitted into the scene.

2.2 Planar shadows

This basic, fast and simple algorithm works with polygonal representation of objects. Shadows are constructed by computing a special transformation for each plane, on which shadows can be cast. This transformation projects every object onto a plane as a flat 2D polygon (i.e. the silhouette of an object is projected onto the plane surface). The construction of a planar shadow is shown in 2.2.

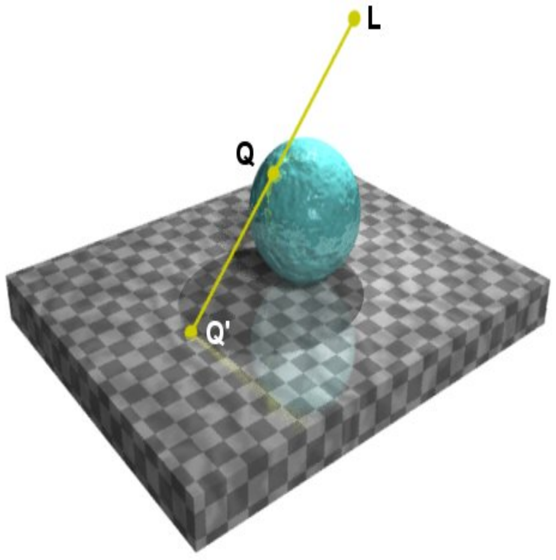

Figure 2.2: Shadow created by a projection of the silhouette of the object onto the plane.

The surface, on which the silhouette is projected, must be a plane, otherwise the algorithm does not work correctly. This method also do not take into account positions of other objects in the scene and therefore it can sometimes produce *fake shadows*. Self-shadowing is impossible as well. Thanks to these properties it cannot be used in applications which require correct shadows.

The algorithm creates hard shadows, but it can be easily modified to generate soft shadows. This can be done by moving a light source a little in a number of render passes and interpolation of the results.

2.3 Shadow maps

Computation of shadows by shadow map method is completely done in an image space. It means that the algorithm do not need any information about scene geometry and the method works with any representation of 3D objects. But it can be used only for point light sources.

The first idea of this algorithm was published by Lance Williams in 1978. Algorithm renders scene twice. In the first pass, the scene is rendered into the *depth-buffer* from the light position. This render creates a *shadow depth map* which is a function mapping the nearest pixel towards the light location into 2D image space.

This map is then used in the second pass, when the scene is rendered from the camera point of view. The map is projected onto the scene and depth of every rendered pixel is compared with the adequate value saved in the map. If the pixel depth is higher, there must be an object that is closer to the light source than the pixel and therefore the pixel is in the shadow.

The main disadvantage of this method is its great memory consumption, incorrectness and alias caused by the limited resolution of depth maps. More detailed description can be found in [15] and [6].

There exist many alternatives of this algorithm, results of two of them shows the figure 2.3. Shadow maps are often combined with *Shadow volumes method* to get fast and robust shadow algorithm.

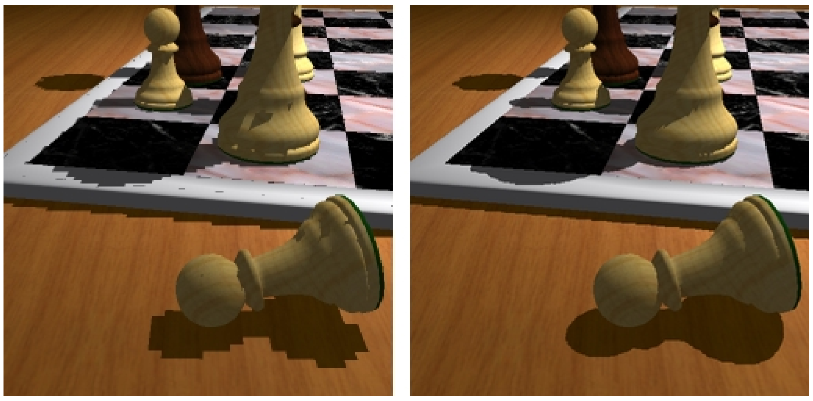

Figure 2.3: Results of two variants of shadow maps algorithm. Left image, with incorrect shadows and not sharp shadow boundary caused by low depth-buffer resolution, is created by a uniform shadow maps. Better results in the right image, which is constructed by perspective shadow maps.

Chapter 3

Shadow volumes method

3.1 Description

This method is widely used, e.g. in Doom 3, to compute shadows in real time thanks to its robustness and hardware support in GPUs. It creates geometrically correct shadows. Unlike other methods it is also able to create self shadows. It allows to create shadows only for selected objects in the scene, which can be useful in many situations.

The technique is based on computing *shadow volumes* created by light rays and occluders in object space. First idea of shadow volumes was presented by Frank Crow [5] in 1977.

This method in its basic implementation works with polygons and point light sources, thus it creates hard shadows only. Consequently in the rest of the paper the point light sources will be assumed. How this method can be extended to create soft shadows is described in [13].

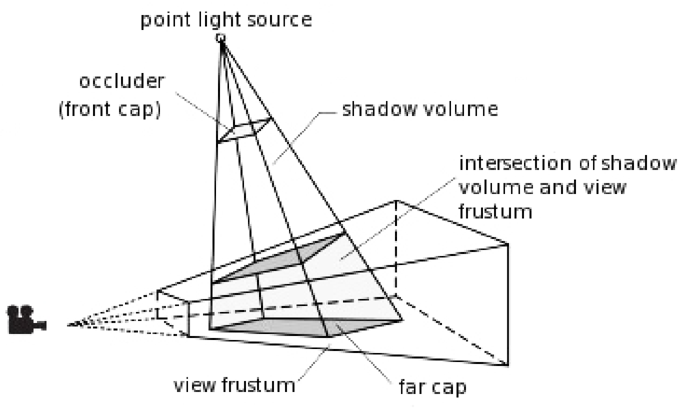

Figure 3.1: Construction of shadow volume and its intersection with view the frustum.

The main idea of this method consists in the construction of *shadow volumes* for each object which can cast shadow. Construction of shadow volume is shown in the figure 3.1. Image shows that the shadow volume sets bounds to space in the scene, where the light source cannot be seen because of the occluder. Every object inside the shadow volume is covered by the occluder's shadow.

The shape of shadow volume is the result of extruding the occluder silhouette edges from the point-of-view of the light source to a finite or infinite distance. Thus, implementations that extrude silhouette edges to infinity are commonly known as *infinite shadow volumes*. Having found all shadow volumes, computing of shadows can be transferred into solving the visibility of scene objects against these volumes. Detail information of shadow volumes technique is described in [10] and [11], or in [4] (czech only).

Construction of a shadow volume for one polygon is not difficult. For occluder of a general shape, its silhouette should be computed first. Silhouette consists of *silhouette edges* and marks the shadow volume shape; each silhouette edge defines one face of the shadow volume.

Determination of the silhouette is either computationally inefficient or hard to implement, but it reduces the number of shadow volumes polygons. Usually the silhouette computation is done in GPU.

However, it is not necessary to compute the silhouette. Shadow volume can be created for every light-facing polygon. That increases the number of shadow volumes rendered polygons and thus slows down shadow rendering.

3.2 Depth-pass algorithm

Determination of visibility of scene objects and shadow volumes is done in raster-space. The idea of computing the visibility is based on the emitting the test rays from the camera through every pixel towards the scene. The figure 3.2 shows that on the way of a ray, it is counted, how many shadow volumes the ray entered and left.

If the difference of these values is not null, it means, that the ray has not left all shadow volumes it has entered. Therefore the pixel on the surface of an object must lie in the shadow.

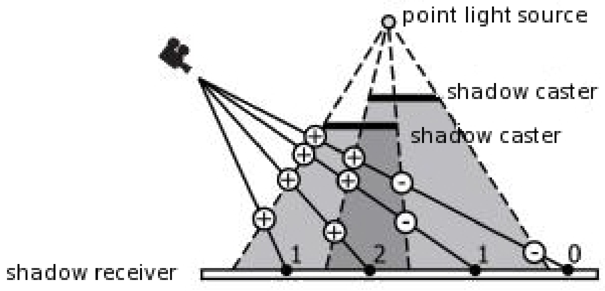

Figure 3.2: The depth-pass algorithm principle. The counter for every pixel (stencil buffer) is incremented at the front faces of shadow volumes if the test passes and decremented on the back faces. Non-zero value at the receiver surface means the pixel is in the shadow.

To determine whether the pixel is in the shadow or not, the depth test is used. At first, the scene without shadow volumes is rendered into the depth buffer and the result is kept in the memory. After that, a counter is added to every pixel, with default value equal to the number of shadow volumes in which the camera is. Then the shadow volumes are rendered. The depth of every pixel of shadow volumes is evaluated against the result kept in the depth-buffer. If the test passes for a pixel, i.e. shadow volume polygon is in front of all objects in the scene, the counter is incremented for front faces of shadow volumes and decremented for back faces. Drawing to frame and depth buffer must be disabled during this operation.

If the counter value is not null after processing all shadow volumes faces, then the pixel is located in the shadow. Hardware stencil buffer can be used to implement the counter, as proposed by Tim Heidmann [9] in 1991.

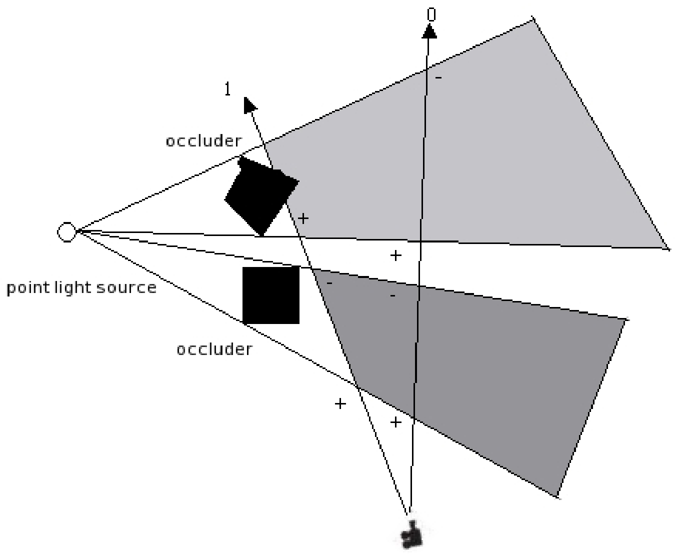

Figure 3.3: The counting using the stencil buffer will still work even for multiple intersecting shadow volumes.

The above algorithm is also known as the *depth-pass* stencil shadow volume technique since the values in stencil buffer are manipulated only when depth test passes. Depth-pass is also commonly known as *z-pass*. The shadow volume counting work for multiple shadow volumes as shown in the figure 3.3.

In practise, it must be taken into account, that some faces of the shadow volume can be clipped by a near or far plane of the view frustum. In that case holes into shadow volume are created, and rendered shadows are not correct. Example of such a situation is in the figure 3.4.

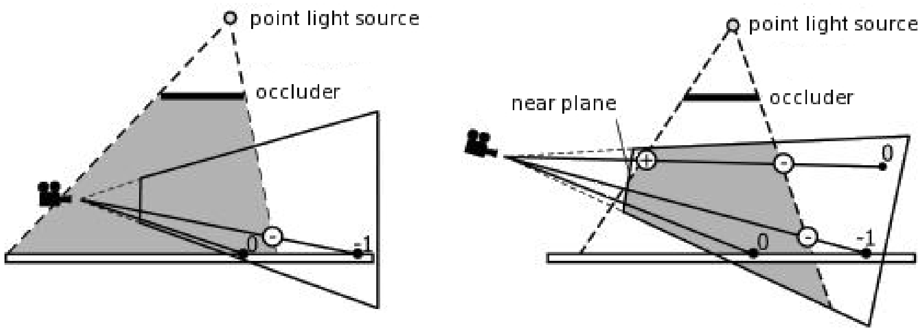

Figure 3.4: Failure of depth-fail algorithm: camera within the shadow volume on the left and shadow volume clipping by near plane of the view frustum on the right image (both lower rays).

3.3 Depth-fail algorithm

To avoid problems caused by the near plane clipping, the reversed sense of counting of visited shadow volumes in depth-pass algorithm is used, i.e. the counter value, initially set to 0, will be updated only if the depth test fails. This way the method counts entered and left shadow volumes with the ray casted from infinity towards the camera. The sense of this method is exactly inverse to depth-pass.

This new algorithm, known as *depth-fail* or *z-fail*, was introduced by John Carmack, Bill Bilodeau and Mike Songy. It solves the near plane clipping and, what is more, it is independent of the original camera position. But there are new requirements compared to *depth-pass*.

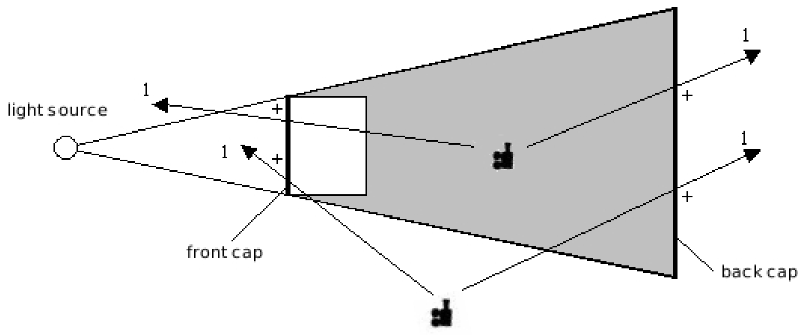

Figure 3.5: Shadow volume capped at the front and back.

To put in non-zero values into the stencil buffer, the depth-fail technique depends on the failure to render the shadow volume back faces with respect to the camera position. This means that the shadow volume must be a closed volume; the shadow volume must be capped at both the front and back end (even if back end is at infinity) as shown in the figure 3.5. Without capping, the depth-fail technique would produce erroneous results.

The front cap can be built by reusing the front facing triangles with respect to the light source. The geometries used in the front cap can then be extruded, but vertices ordering must be reversed, to create the back cap. Reversing the ordering is to ensure that the back cap faces outwards from the shadow volume.

To avoid far plane clipping, there is a need of infinite shadow volumes. However, it is not compulsory to extrude the silhouette edges to infinity, if it is certain that the shadow volumes cover all objects that are in front of the view frustum far plane. In practical cases, a large value would normally be more than adequate.

In fact, it must be always found out that the primitives, in our case triangles defining the entire shadow volume are outward facing (their normals must be set up correctly) as shown in the figure 3.6.

Rendering of closed shadow volumes are somewhat more expensive than using depth-pass without capped shadow volumes. Besides a larger primitive count for the shadow volume, additional computational resource are also needed to compute the front and back capping. Thus a number of optimization techniques were developed, more in [7].

It must be pointed out that neither the depth-pass nor depth-fail techniques are perfect. In fact, combination of different methods is needed to come up with a robust solution for shadow volumes. Often both algorithms are implemented at once and as lately as during the rendering of the scene a decision is made, whether to use simple and faster depth-pass or slower, but robust,

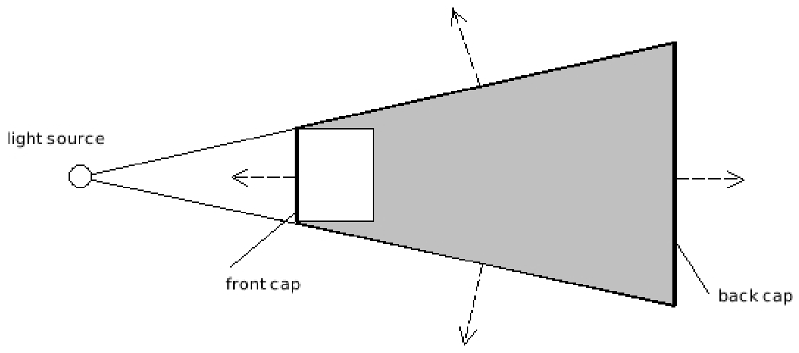

Figure 3.6: Normals orientation of shadow volume faces.

depth-fail algorithm, depending on the camera position. For an example of shadow volumes, see the figure 3.7.

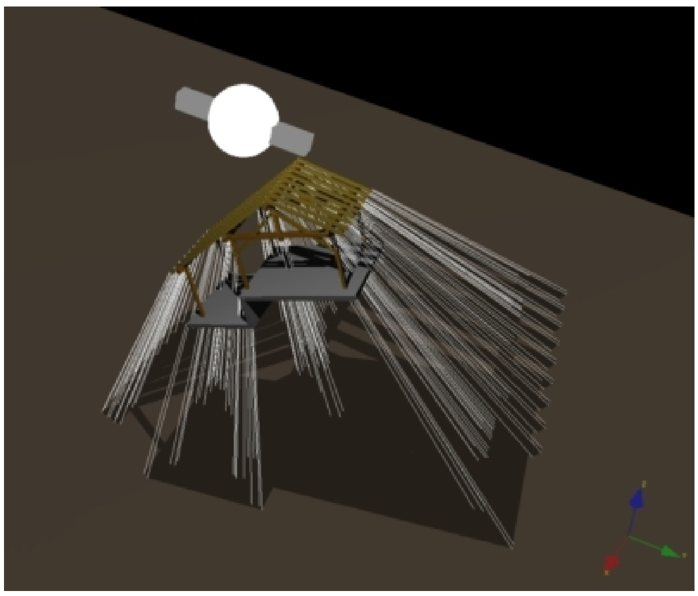

Figure 3.7: Shadow volumes of a wooden construction. [Shadow Engine]

3.4 Shadow volumes rendering

The rendering algorithm for a single frame runs through the following steps.

1. Clearing the frame buffer and performing an ambient rendering pass. Rendering the visible scene using any surface shading attribute that does not depend on any particular light source.
2. Choosing a light source and determining what objects may cast shadows into the visible region of the world. Clearing the stencil buffer.
3. For each object, calculating the silhouette and constructing a shadow volume by extruding the silhouette away from the light source.

4. Rendering the shadow volume using specific stencil operations that leave nonzero values in the stencil buffer where surfaces are in shadow.
5. Performing a lighting pass using the stencil test to mask areas that are not illuminated by the light source.
6. Repeating steps 2 through 5 for every light source that may illuminate the visible region of the world.

For a scene illuminated by n lights, this algorithm requires at least n+1 rendering passes. To efficiently render a large scene containing many lights, one must be careful during each pass to render only objects that could potentially be illuminated by a particular light source.

Chapter 4

The Shadow Engine

4.1 Need of a shadow engine

As mentioned before, there has not been any possibility of shadows in Open Inventor recently. Therefore, this project was founded to develop a programmer library with simple API, that would use one of shadow generating methods to enrich Open Inventor applications of shadows.

For its good properties and robustness, the *Shadow volumes method* has been chosen, but it is possible to implement also other algorithms in the future.

Before the design and requirements of the Shadow Engine will be discussed, the basic structure of Open Inventor must be described.

4.2 Open Inventor

Open Inventor by SGI is a high-level 3D graphics toolkit for developing cross-platform real-time 3D visualization and visual simulation software. This project uses free-software version of Open Inventor called *Coin3D* [1] developed by Systems In Motion, Norway. Coin3D is fully compatible with SGI Open Inventor 2.1 and it is released in two versions: Professional Edition for the development of proprietary software, and a Free Edition under the GPL for Free Software development.

Coin3D is written in C++ language and built on OpenGL. It uses *scene graph* data structures to render 3D graphics in real-time. Thus the development of an application is easier and faster compared with programming directly with OpenGL. OpenGL code and Coin3D code can co-exist in the same application. Good tutorial about Coin3D can be found in [12] (czech only). For more information about Open Inventor API read [14].

Scene graph, which describes 3D scene, is in Open Inventor represented by a tree-like structure, which can consist from many node types. Basic node types are those that describes geometry of an object (coordinates, normals and indexes for indexed geometry types), nodes which hold information about object properties (e.g. material). Some special nodes define object transformations (translation, rotation, scale, etc.), sensors, timers, cameras, etc. There are also nodes that group other ones (e.g. separators) and build the tree structure in that way.

Rendering of a scene is done through the traversal of a created scene graph from its root node through child nodes from left to right to leafs of the tree. Thanks to this structure, it is possible to search for particular nodes (even by its name), add callback actions that are fired off when a certain node is traversed.

Rendered scene can be shown, besides the basic render area, in special viewers, which give the user some extra functionality for viewing and exploring the 3D scene. These viewers are contained in two libraries, SoWin and SoQt.

4.3 Engine design

The most important feature of the Shadow Engine is generation of correct object shadows. Beside of that, it must detect any light and object movements in the scene and recompute shadows as well as force shadow recomputing when any object changes shape. This all must be done fully automatically, without any user action.

An user should only define for which lights and objects shall shadows be created, set the engine, and show the final scene graph with shadows in a viewer.

The main requirements of the Shadow Engine are:

- Simple API, easy usage.
- Registering shadow casting objects and lights for which shadows should be computed.
- Shadow computation independent of object geometry type.
- Shadow computation for all light types of Open Inventor.
- Generation of geometrically correct shadows.
- Generation of shadows for multiple light sources of different colours.
- Shadow update when object moves.
- Shadow update when object changes its shape.
- Shadows update when light source moves.
- Real-time functionality.
- Open Inventor compatibility.

Shadow Manager

As in every engine, there must be a main control unit in the Shadow Engine as well. It is called *Shadow Manager* and forms the interface of the whole engine with user. It hides the functionality of the Shadow Engine and implementation details. It is the only class which should user create in his application and communicate with.

The figure 4.1 shows the most important ways, how a user can communicate with Shadow Manager. As shown in the picture, a user must register in Shadow Manager all lights and objects from his scene graph, for which shadows should be computed. De-registration of lights and objects from manager is possible as well.

The root of user scene graph must be also known in the manager to search for information in it (e.g. paths to nodes, environment node, etc.). Shadow Manger has a number of parameters which affect shadow computation. User can adjust them to suit his scene through the settings.

When all lights and scene objects are added to the Shadow Manager and the manager is set, user can ask it for the root node of his scene with shadows, which is then shown in the scene viewer.

This is the most important output of the Shadow Manager, another one is, for example, export of the scene with shadow volumes into a iv file.

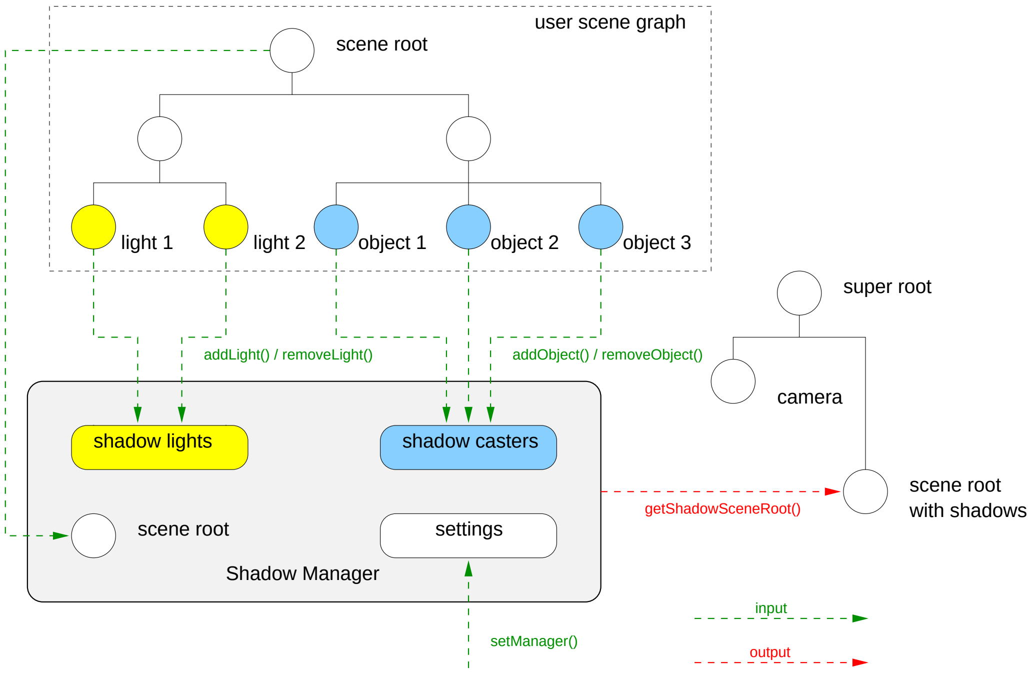

Figure 4.1: The main inputs and output of Shadow Manager. User can register and deregister scene lights and objects in Shadow Manager and set the manager. The root of the user scene graph must be known to the manager as well. After that, user can ask the manager for the root node of scene with shadows.

Shadow scene graph

As mentioned before, Shadow volumes method computes *shadow volumes* to render shadows. These volumes are common scene objects as another ones. The only exception is, that they are used only in rendering to create stencil of shadowed areas. They are not visible in normal situations.

Because Open Inventor renders scene by traversing the scene graph, shadow volumes must be placed in this scene graph as well. Implementation details will be discussed later.

Shadow volume must be computed for every light and every shadow casting object, called *caster* shortly. It is assumed that all lights enlighten all objects in the scene for simplicity[1]. For each volume a node that holds its coordinates and normals is created.

There are two possible ways of how to organise these shadow volume nodes in a graph. The first one, it can be called *object oriented*, builds up the graph according the scene objects.

A new node is created for each shadow caster and shadow volume nodes for all lights are appended to it. For an example of such a graph see the figure 4.2.

This schema was used in early versions of the Shadow Engine, which was not able to draw correctly coloured shadows for more than one light source. Its advantage is that it is easy to manipulate (add and remove) shadow casters. When a new caster is added and its node is created, shadow volume nodes of all lights are appended directly to it. This seemed to be handful, because scene objects are manipulated more often than lights. When an object moves, it is also easy to find and recompute its shadow volumes, because they are children of the caster.

[1] For example, if there are 2 lights and 3 scene objects, it means, that 6 shadow volumes must be computed.

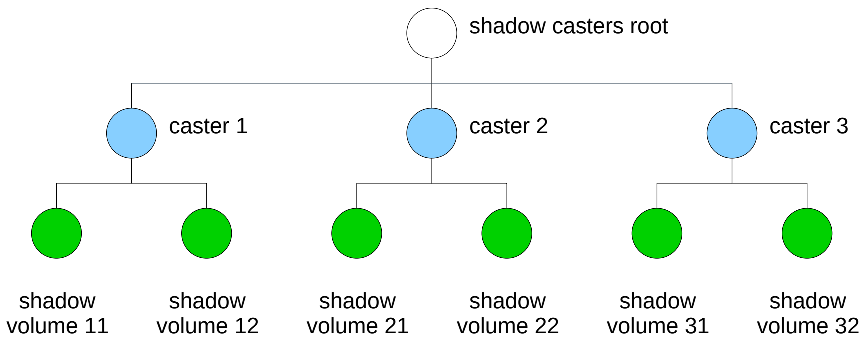

Figure 4.2: Object-oriented scene graph of shadow volume nodes.

The other way is to create shadow volumes scene graph based on scene lights. For each light a new node is created and shadow volume nodes of all objects, that are enlightened by this light, are appended to it. Such a graph shows the figure 4.3.

Adding and removing of scene object is in this case a little bit more complicated, as a shadow volume node must be appended or disconnected at a right place (proper shadow volume node must be found first).

This causes a small delay when an object moves, because all its shadow volumes, which are appended to different light nodes, must be found and recomputed.

But the light-oriented scene graph proved as a need for rendering coloured shadows for multiple light sources as the algorithm needs to render all shadow volumes of one light together.

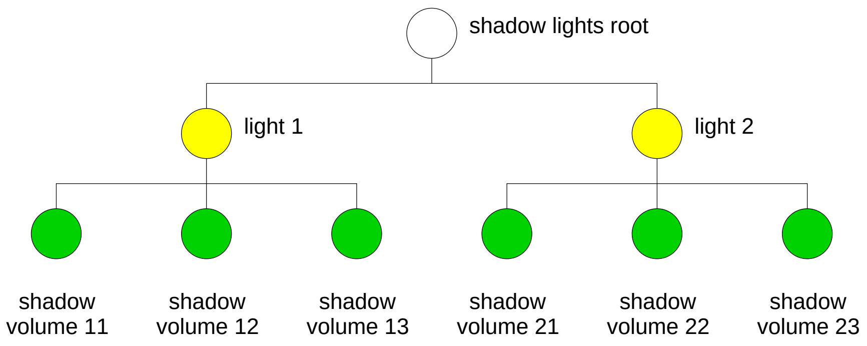

Figure 4.3: Light-oriented scene graph of shadow volume nodes.

Shadow casters

A shadow caster was supposed to be any scene object by now. But what is the meaning of shadow casting object in Shadow Manager? In Open Inventor it is possible to separate the whole scene graph into sub-trees. One sub-tree can, for example, be a model of a car rooted at `SoSeparator` node 'car'. The sub-tree contains information about material of the car, coordinates, normals, transformations, etc., and defines the whole car.

The root node of this tree, 'car', represents a complete object in the Shadow Manger. the manager like this:

```
shadowManager->addObject(car);
```

Assume, that in the user scene, there are this five cars parked at a parking place. This can be easily done in Open Inventor by appending the ‘car’ node to some root, appending a translation node to the same root, which moves global coordinates to the place of the second car, and appending the ‘car’ node again. This can be done many times.

If the user want these cars to drop shadows, he would call `addObject(car);` method for each of his cars’ nodes. If the user wants to remove shadow of the third car, for example, he would call `removeObject(car);` as expected. But which ‘car’ node of the five registered should be removed, when they are the same?

Therefore it is possible to register each node in Shadow Manager only once. If the user wants to add shadows to all five cars, he must create unique node for each car, append a translation (or other transformations to it) and append a ‘car’ node to it. When registering cars to Shadow Manager, user must register these unique nodes, under which each car is placed.

Chapter 5

Implementation

5.1 Implementation introduction

Shadow Manager defines the API of the Shadow Engine and manages all user actions. Its structure is defined in the class `CShadowManager`. To create shadows, some other developed classes are needed.

Each class name has some prefix. Prefix ‘C’ means that it is an ordinary class, whereas ‘So’ means ‘scene object’, as common in Open Inventor. Classes, which name begin with ‘So’, are derived from `SoSeparator` and therefore they can be appended to other nodes in a scene graph, and other nodes can be appended to them as well.

Class `SoShadowCaster` defines the data structure of a shadow casting object, `CObjectModel` is its model. This model must be computed in order to be able to compute shadow volumes independently of object polygon type.

Classes `SoShadowLight` and `SoShadowVolume` represent shadow light and a shadow volume nodes, that build up the scene graph with shadows.

As shown in the class diagram in the figure 5.1, it is possible to access all information through the Shadow Manager class. Therefore Shadow Caster and Shadow Light classes have pointers to it.

5.2 Shadow Manager class

As mentioned before, Shadow Manager is the most important class in the whole engine. There must be defined two nodes in its constructor. The first one is the root of the user scene graph and the second one is an environment node.

Environment node

The environment node controls global ambient light colour and intensity and it can also set fog and other natural effects. It is assumed, that this node is in the user scene graph only one, if any. If there is no environment node, user can set the second parameter to NULL. Shadow Manager will then try to search for this node in the graph and uses it, if it is found. If not, the manager creates its own environment node, because it is necessary for controlling the ambient light intensity, which defines shadow colour.

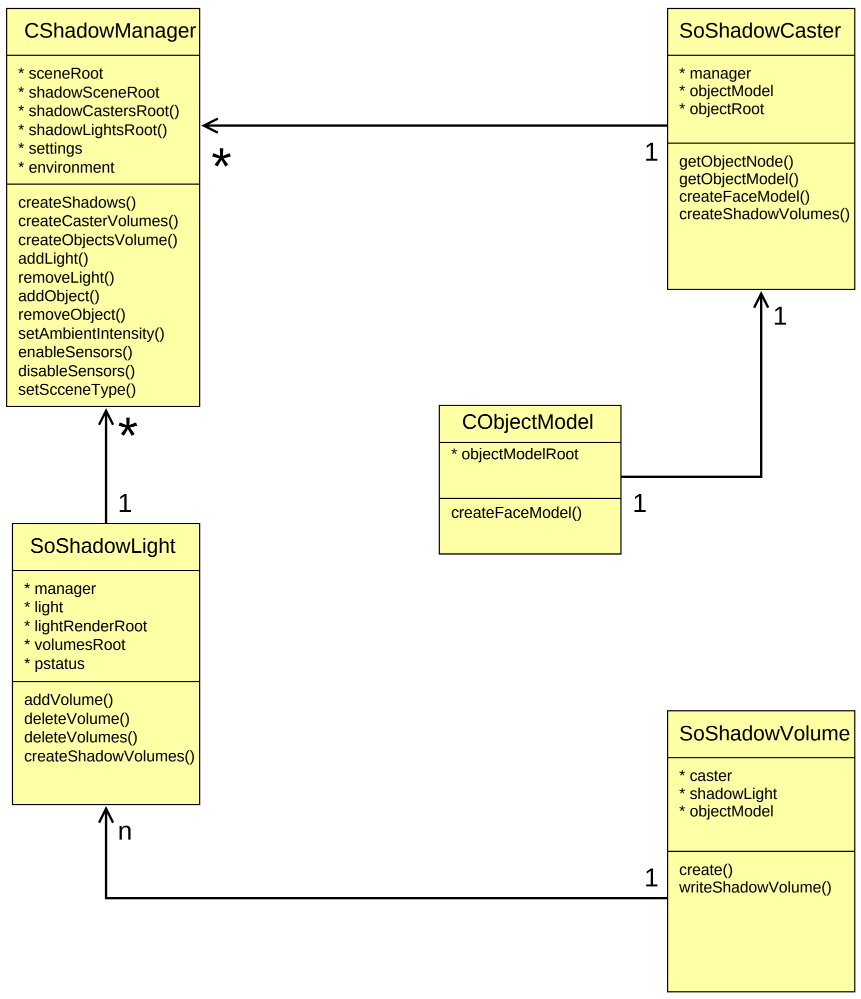

Figure 5.1: The Shadow Engine class diagram. Only the most important attributes and methods are shown in the picture.

Shadow light nodes

The main task of Shadow Manager is to register lights and objects, for which shadows should be computed. Therefore there exist two root nodes: one for shadow lights and the other for shadow casters.

When a light is added to the manager, a new instance of `SoShadowLight` is created and appended to the ‘shadowLightsRoot’. New `SoShadowVolume` nodes are created for registered shadow casters and appended to the new shadow light node. Computation of shadow volumes is forced for the first time, after that.

To be able to update shadows when this light moves, it is necessary to create a sensor attached to the light node. When it detects a light movement, it runs a callback function ‘lightMoveCallback’, which causes recomputing of shadow volumes for this light and all registered objects.

Shadow caster nodes

A similar process is followed when a new shadow casting object is to be registered in the Shadow Manager. At first, it is checked whether the same node had not been added to the manager before. If not, a new `SoShadowCaster` node is created and new shadow volume nodes are created for all existing lights and appended to the right shadow light nodes. Shadow volumes of this caster are computed for the first time. To create shadows of an object, its model must be computed first, as it will be explained in 5.3.

When an object moves, its shadows must be recomputed. It means that a sensor, which causes recomputing of shadows of the object, must be attached to the object node. But in the case of object, shadows also change when the shape of the object changes (e.g. the scale of the model increases or a car looses one of its wheels).

Therefore there must be a second sensor that detects any changes in the object geometry and in the path from the root of the scene to the node of the object (to catch inserting of some transformation node before, for example).

Sensors in Open Inventor detect any change, but they can be adjusted to react only to defined node type changes. This feature is used in the Shadow Manager callback functions.

Setting the Shadow Manager

Shadow engine implements both z-fail and z-pass algorithms of Shadow volume method. User can set, which algorithm shall be used by setting appropriate method. It has three possibilities: zpass, zfail and autoset. The last one is not yet implemented, but it will be used to determine the shadow algorithm according the position of objects against light when the application is running, which can speed up rendering.

User can determine, whether scene shall be rendered with or without shadows through 'setSceneType'. The complete shadow scene graph is shown in the figure 5.2.

Shadow scene graph

In this graph, switch node is used to switch between sub-graphs of the scene with shadows and without shadows. The third light-red node in the graph should show scene with visible shadow volumes, but it is not implemented yet.

Dashed environment node is connected to this graph, only if it is created by the Shadow Manager itself. The square objects are callback functions which change OpenGL state machine during rendering.

The sub-graph under each shadow light node will be described in section 5.5.

All objects of the shadow scene graph, shadow lights, casters and shadow volumes, have their unique names, which are used, when a some node is searched for.

Shadow Manager can also set the depth of shadow volumes and export shadow scene graph into a file.

5.3 Object model class

In order to construct shadow volumes independently of object geometry type (SoFaceset, SoIndexedTriangleStripset or other), a method that can operate on all geometry types must be used. Such a method is `SoCallbackAction` in Open Inventor. After adding an `addTriangleCallback` to it, this method can call a user specific callback function for every triangle it finds.

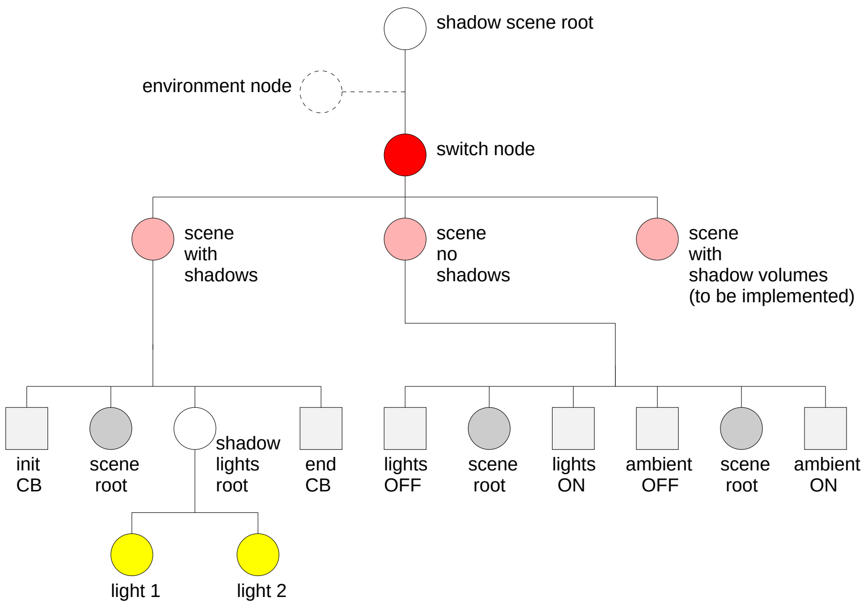

Figure 5.2: Top part of the shadow rendering scene graph. Switch node is used to set, whether scene is rendered with or without shadows. The square objects are callback functions which change OpenGL state machine during rendering.

Using this ability, a function, that transforms every triangle of an object to one geometry type, can be called. Having computed the object model, only one algorithm working on this geometry is needed. Object model is exactly the same as the original model except for transparent parts. These do not cast shadow (because they are transparent) and therefore must be removed from object shape. Example of such an object and its model is to be found in the figure 5.3.

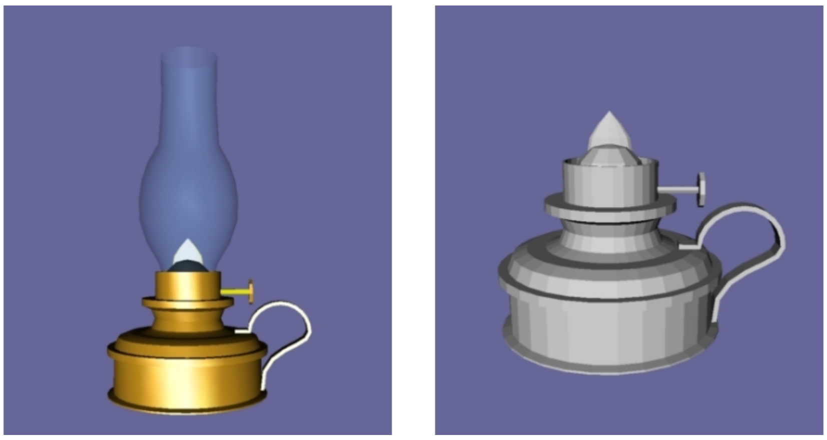

Figure 5.3: A lamp with transparent lampshade (on the left) and its model without transparent triangles (on the right). Transparent parts do not cast shadows and therefore they are let out from the model.

Computation of the object model is the purpose of `CObjectModel` class. Each shadow caster has a method `createFaceModel` that creates an object model consisting of `SoIndexedFaceSet` by applying a `TriangleCB` callback action to its root node.

Every triangle of the object is converted into `SoIndexedFaceSet` and its vertices are transformed into global space coordinates. Normal vector is computed for each triangle here (counter-clockwise triangle orientation is assumed). Texture coordinates are not important, material of an object is used to filter transparent triangles.

As for callback functions seem to work slowly, model is computed only once and then kept in the memory. But if the geometry of the object changes, e.g. a wing of a plane falls away, the model must be recomputed.

5.4 Shadow caster class

Shadow caster class represents a shadow casting object. When an object is added into the Shadow Manager, an instance of this class is created and appended to the 'shadowCastersRoot root'.

There is no need to keep shadow casters in any hierarchy of nodes, but the reason, why this class is derived from `SoSeparator` is, that it simplifies searching for a certain node.

Each shadow caster has an information about which scene object it belongs to, so it is possible for the user to force the Shadow Manager to recompute shadows of this object (for example when sensors do not detect some change of the object).

Another pieces of information, which hold this class, is its name, pointers to object sensors and transformation matrix, and the pointer to the computed object model.

5.5 Shadow light class

As mentioned before, the shadow scene graph is based on `SoShadowLight` nodes to render correctly coloured shadows by multiple light sources. When a new light is added to the Shadow Manager, shadow light node is created and appended to the shadow scene graph, whose top part is to be shown the figure 5.2.

Each shadow light node is the root node of a sub-graph for rendering shadows cast by this light. Its structure is shown in the figure 5.4. There are four callbacks in the graph, which purpose will be explained later.

Polygon offset node

The *polygon offset* node must be inserted into the graph because of the limited resolution of depth buffer, which affects rendering of triangles at the same depth. Without this node, artifacts on objects could be seen in depth-fail algorithm, which demands the shadow volumes to be capped (the near cap of the shadow volumes is exactly at the same depth as the triangle of an object). Setting of values of polygon offset node is very important.

A shadow light sub-graph

As shown in the image, node named 'volumes root' is the root of all shadow volume nodes, which are created for all shadow casting objects. This node is appended to the 'light render root' twice.

The root of the user scene must be appending to 'light render root' as well, to render the enlightened areas by the light.

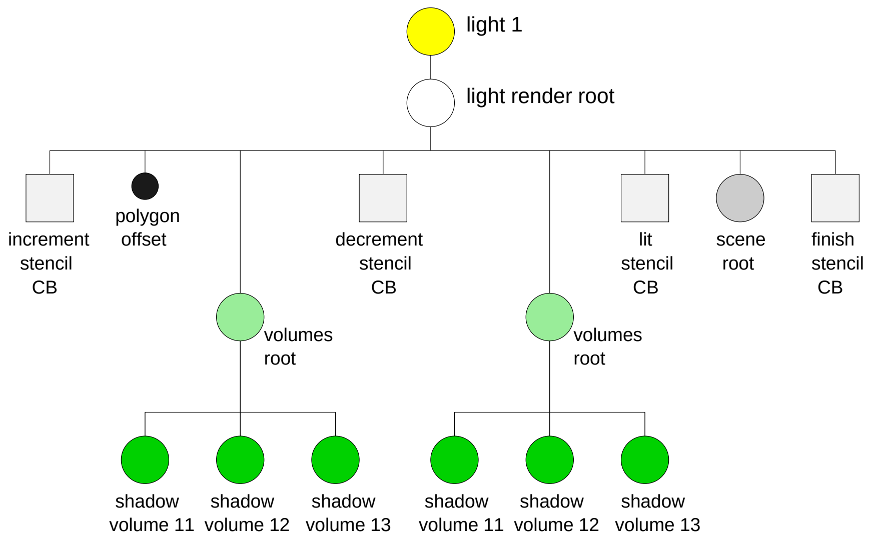

Figure 5.4: The sub-graph of shadow rendering scene for one light.

Implementation of shadow volumes rendering

Rendering shadow volumes was described in section 3.4. In Open Inventor it is implemented as a scene graph, to which callback functions are appended. These callbacks change the settings of OpenGL state machine, which is needed to render shadows by Shadow volumes method.

Now, the purpose and important pieces of code of shadow rendering callbacks shall be explained now. Two of them, 'initCB' and 'endCB', are implemented in the Shadow Manager class, because they start and end the whole rendering loop for all lights.

initCB Sets initial OpenGL rendering state and clears all buffers. Turns off all lights (and saves their state values) to render the user scene (which is appended as the next node in the figure 5.2) only with ambient light. Depth buffer is enabled to initialize its values.

The loop for each shadow light follows. These callbacks are placed under each light to render its shadows.

incrementStencilCB The stencil buffer is cleared and the stencil test is configured to pass always. The depth test is set to pass only when fragment depth values are less than those already in the depth buffer.

Writing into the depth buffer and the colour buffer is disabled, initialization of stencil values is needed only.

The stencil operation is set according to the used algorithm to modify the values in the stencil buffer when the depth test passes, if *depth-pass* algorithm is used. The stencil value is incremented for fragments belonging to front-facing polygons.

In the case of *depth-fail*, the values in the stencil buffer are modified when the depth test fails. The stencil value is incremented for fragments belonging to back-facing polygons.

```
...
glClear( GL_STENCIL_BUFFER_BIT );
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glShadeModel(GL_FLAT);
glDepthMask(GL_FALSE);
glDepthFunc(GL_LESS);
glEnable(GL_STENCIL_TEST);

if ( algorithm == zfail ) {
  glStencilFunc(GL_ALWAYS, 0, ~0);
  glStencilOp(GL_KEEP, GL_INCR, GL_KEEP);
  glCullFace(GL_FRONT);
} else {
  glStencilFunc(GL_ALWAYS, 0, ~0);
  glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
  glCullFace(GL_BACK);
}
...
```

After this callback, the 'volumes root' node with all shadow volumes of one light is appended to be rendered using above settings..

decrementStencilCB If *depth-pass* is used, decrement stencil values for fragments belonging to back-facing polygons, otherwise decrement stencil buffer values for fragments belonging to front-facing polygons (the opposite of the depth-pass operations), when *depth-fail* is used.

```
...
if ( algorithm == zfail ) {
  glStencilOp(GL_KEEP, GL_DECR, GL_KEEP);
  glCullFace(GL_BACK);
} else {
  glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);
  glCullFace(GL_FRONT);
}
...
```

Render the shadow volumes again.

litStencilCB Once shadow volumes have been rendered for all objects, a lighting pass that illuminates surfaces wherever the stencil value remains zero is performed.

Writing to the colour buffer must be re-enabled, the depth-test changed to pass only when fragment depth values are equal to those in the depth buffer, and the stencil test is configured to pass only when the value in the stencil buffer is zero.

Since the lighting pass adds to the ambient illumination already present in the colour buffer, blending equation must be used.

The ambient light is turned off and the light, whose shadows are rendered is turned on.

```
...
// turn off ambient light

// turn on the selected light, whose shadows are rendered

glEnable(GL_LIGHTING);
glBlendFunc(GL_ONE, GL_ONE);
glEnable(GL_BLEND);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthFunc(GL_EQUAL);
glStencilFunc(GL_EQUAL, 0, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glEnable(GL_STENCIL_TEST);
...
```

User scene is add after this callback to be rendered.

finishStencilCB Turns on the ambient and turns off light, whose shadows were rendered. Stencil and blending operations are disabled as well as lighting.

```
...
glDisable(GL_LIGHTING);
glDisable(GL_STENCIL_TEST);
glDisable(GL_BLEND);
...
```

End of the loop for each shadow light.

endCB Closes the whole rendering cycle, sets all lights states to their saved values.

Rendering problems

During implementation of rendering shadow volumes two problems were faced.

Since shadow volumes are constructed for every light-facing triangle of the silhouette, in shadows of models with large number of triangles a problem appeared, that there was a not any shadow at the place where it was expected. This situation is shown in the figure 5.5, left image.

This problem was caused by overstepping of the highest value that can be put in the stencil buffer. Stencil buffer has usually 8 bits, and when there was a large number of shadow volumes rendered at one pixel, the maximal value was not enough, the higher values were cut off.

This is fixed with using of the GL_INCR_WRAP_EXT and GL_DECR_WRAP_EXT stencil operations instead of GL_INCR and GL_DECR. These are provided by the GL_EXT_stencil_wrap extension to OpenGL and allow stencil values to wrap when they exceed the minimum and maximum stencil values instead of being clamped.

A static method 'checkStencilExt' of `SoShadowLight` checks if this OpenGL extension is available and uses it, if possible.

The second problem concerned of rendering of the scene with and without shadows. Usage of blending function in the rendering of the shadowed scene caused, probably, that scene seemed to be

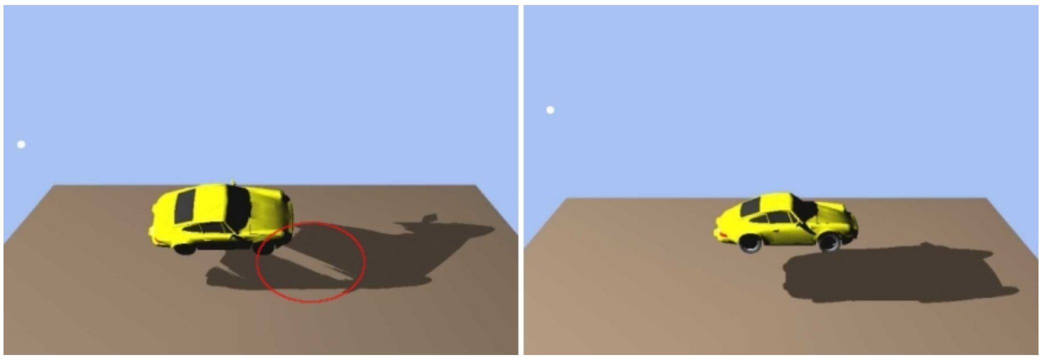

Figure 5.5: Incorrect shadow of high-detailed model of a porsche caused by limitations of the stencil buffer (left image). Problem area is highlighted, but incorrectness is visible also in other parts of the shadow. Right image shows shadow computed by fixed algorithm using GL_EXT_stencil_wrap extension. [Shadow Engine]

more lighted than the same scene without shadows. This was notable when shadows were turned on and off.

Therefore, blending function was used in the scene without shadows as well and the scene is rendered in two passes now. The first is the ambient pass, when user scene is rendered with ambient light only.

The second pass is the light-pass. The scene is rendered with ambient light turned off and all lights (which are enabled) turned on, and the blending function is configured as in shadows rendering. The figure 5.2 shows scene graph used for rendering of a scene without shadows.

5.6 Shadow volume class

This class is a template for nodes, that compute and keep the shadow volume geometry for one caster and one light. Since silhouette determination is not implemented in the Shadow Engine, shadow volumes are computed for every light-facing triangle. Vertex coordinates, normals and indexes of shadow casting triangles are appended directly to the shadow volume node.

The class most important method called 'create' deletes possible nodes appended to the shadow volume node and recomputes new shadow volume consisting of shadow volumes of every light-facing triangle. In the future, it can be re-implemented to compute silhouette of the object first and make the geometry of the shadow volume much simpler.

All information to compute the shadow volume, i.e. object model, light source and used algorithm, is reachable for this node, because it knows to which shadow light node and shadow caster it belongs.

There are three main types of lights in Open Inventor, point light, spot light and directional light. Computation of shadow volumes is different for each. Spot lights are not used so often, therefore, for simplification, they are treated as point lights.

The main different between a point light source and a directional light is, that point light has exact location whereas a directional light is placed somewhere in the infinity and is defined through its light direction vector.

Determination of light-facing polygons

As mentioned before, shadow volumes are computed for every light-facing triangle. Therefore these must be found in the object model first in order to determine an occluder.

To tell, whether the triangle is light facing or not, computation of the dot product of its normal and a 'light vector' must be done.

For a point light source, the 'light vector' can be defined as a vector oriented from one its point to the light source. But in the case of a directional light it is the opposite vector to the light direction vector.

If the result of the dot product is positive, $\vec{n}_a \cdot \vec{s}_a > 0$, i.e. the angle between the normal and light vector of the triangle is less than 90 degrees as shown in the figure 5.6, the triangle is light facing and it is part of the silhouette, otherwise it is not. The edge shared both light-facing and not light-facing triangles is a *silhouette edge*.

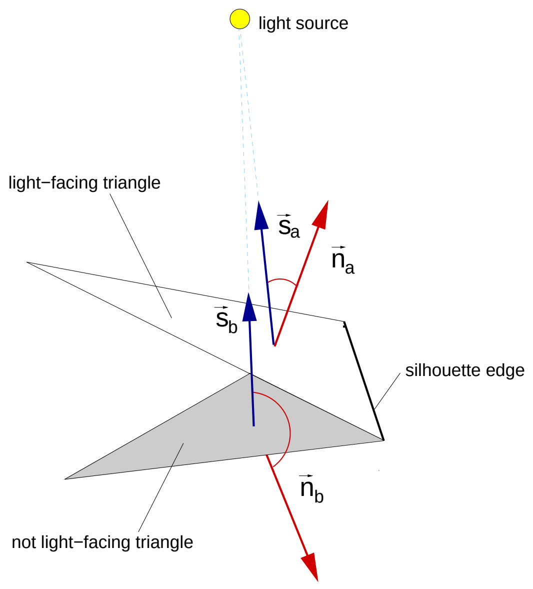

Figure 5.6: Determination of light facing triangle. Normal vectors are of red colour, light vectors are of blue colour.

Shadow volume construction

Shadow volume is created for every light-facing triangle by casting a ray originating in the point light source towards to each of its vertices. In the case of a directional light, these rays are parallel to the light direction vector and go through the each vertex of the light-facing triangle.

If these rays are extended to infinity, the shadow volume is created. Actually, the ray do not have to be extended to infinity, a high value is adequate for most applications (the length of rays sets shadow volume depth). But care must be taken so that the shadow volume is not clipped by far plane of the view frustum.

Shadow volumes created this way are good enough for z-pass algorithm. To use depth-fail, caps must be added to the volume at the front and back end. The front cap can be formed by the light-facing triangles, the back cap by their extruding to the volume depth. The figure 5.7 shows the construction of a shadow volume for both point and directional light types.

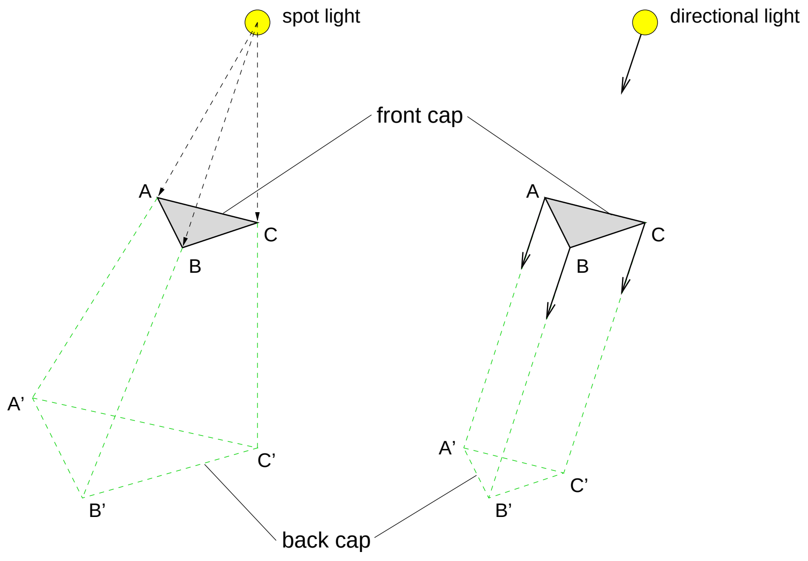

Figure 5.7: Shadow volume construction for a point light source on the left and for directional light on the right.

Normal vectors of triangles of the shadow volumes must be oriented outward according to the figure 3.6 otherwise stencil buffer is not updated correctly.

If a shadow volumes of all object light-facing triangles is computed, it can be considered as the shadow volume of an object (its model).

Example of a shadow volume of a chair is in the figure 6.1.

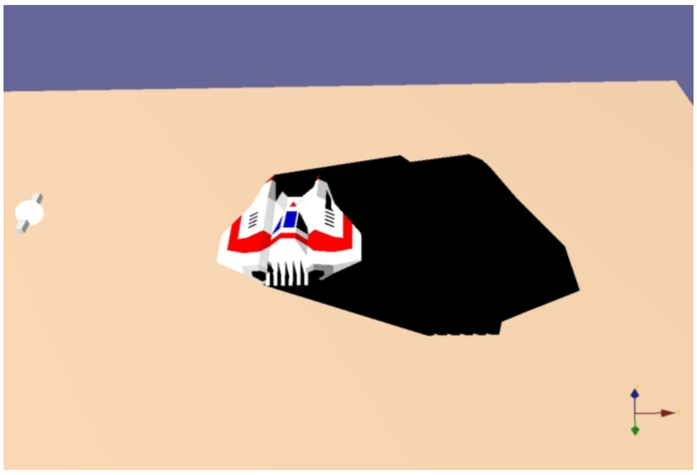

Figure 5.8: Shadow volume of a space fighter. [Shadow Engine]

5.7 Optimalization

Computation of the shadow volume for every light-facing triangle takes a lot of CPU time, especially for detail models with large number of polygons. These shadow volumes have also more triangles than a one shadow volume for a silhouette of an object. Rendering of a higher number of polygons is also slower.

But the computation of a silhouette is not fast as well and it is not trivial to implement. Usually, it is done directly in the graphic hardware.

As mentioned above, shadow volumes must be recomputed when either light has moved or casting object has moved or its shape has changed. As for as directional light types, it is not necessary to recompute shadow volumes when *shadow casting object moves*, because the location of the light is in the infinite distance and its shadow volumes change only when the direction of the light direction vector changes.

There is a small optimalization in the Shadow Engine. Shadow volumes of directional lights are not recomputed when objects are moving, they are only translated to the new object positions. This is much faster than computing of the shadow volume.

Since directional light, which represents sun, is often used in computer games (and the sun does not move so often as objects). This nice feature can make shadow rendering faster.

Chapter 6

The Shadow Engine usage

6.1 Shadows by the Shadow Engine

Shadows generated by the Shadow Engine library can be easily included to Open Inventor applications to add more reality into them. API description for programmers and all sources with some examples can be found among other Open Inventor projects [3] developed at the Brno University of Technology, Faculty of Information Technology.

The Shadow Engine is now used by *LexoView* [2], application to view Open Inventor and CAD models with many features. Hopefully, the number of applications using the Shadow Engine will increase.

During the development of the Shadow Engine, there was found a bug in SIM Coin3D library, version 2.4.4. When a directional light manipulator was moved in the scene, the application crashed, because the class `SoDirectionalLight` was not initialized properly. This bug was fixed in a new version 2.4.5. All examples on the included CD are compiled with this new version of Coin3D.

6.2 Compilation and usage of the Shadow Engine

The Shadow Engine was developed and tested under MS Windows, but it should run under Linuxes as well. It is possible to use it with both SoWin and SoQt GUI libraries.

Following steps must be taken to incorporate and compile the Shadow Engine in a MS Visual Studio application project.

- Download and copy ShadowEngine package into the application code directory.
- In the code include Shadow Manger by

```
#include ‘‘ShadowEngine\ShadowManager.h’’
```

- Change the MS Visual C++ project settings to link the application with glu32.lib opengl32.lib (add these libraries because Shadow Engine uses bits of OpenGL).
- Add Shadow Engine files to your MS Visual C++ project.
- Create CShadowManager instance.

```
CShadowManager * shadowManager = new CShadowManager(
                                     SceneRoot,
                                     EnvironmentNode);
```

SceneRoot is the pointer to root node of the complete scene.

EnvironmentNode is the pointer to SoEnvironment node. The darkness of the shadows is controlled trough this node by setting the 'ambientIntensity'.

- Enable the stencil buffer by yourself. Otherwise any shadows will not be seen.

```
viewer->setStencilBuffer(TRUE);
```

- Add lights to the Shadow Manager.
- Add objects to the Shadow Manager.
- Ask Shadow Manager for the scene root node with shadows.

```
shadowManager->getShadowSceneRoot();
```

- That's all. Shadows should be seen in the scene, if there were no errors.

The code of the Shadow Engine minimal application example can be found in the appendix 10.

6.3 Shadows demo application

The application 'shadows_demo' has been created to test the Shadow Engine properties. User can load an Open Inventor model from a file and place it into a simple scene, where its shadows are casted. It has been compiled under MS Windows.

To load an Open Inventor file (iv or wrl), use '-f filename.iv' command line parameter. To see all command line settings write 'shadows_demo -h'. When a model is opened in the 'shadows demo', it is possible to zoom it in or out by *F5* and *F6* keys.

After the application starts, a new window (`SoWinExaminerViewer` of Coin3D's SoWin library) with the scene is opened. The scene can be controlled in two modes. In the default mode, user can rotate the scene by mouse and move the viewer's camera. If *S* key is pressed, the camera view is zoomed and centered to the point, where is clicked by a mouse.

By pressing *ESC* key, the application changes for action mode. Here user can move the object by arrow keys and move light sources by mouse and thus change the shadows of objects. In this mode, user can use action keys as written in table 6.3.

It is possible to manipulate with the model, insert and move lights of all types in the scene, set light intensity, switch between the depth-pass and the depth-fail algorithms, etc.

By pressing *ESC* again, the application goes back to default viewer mode.

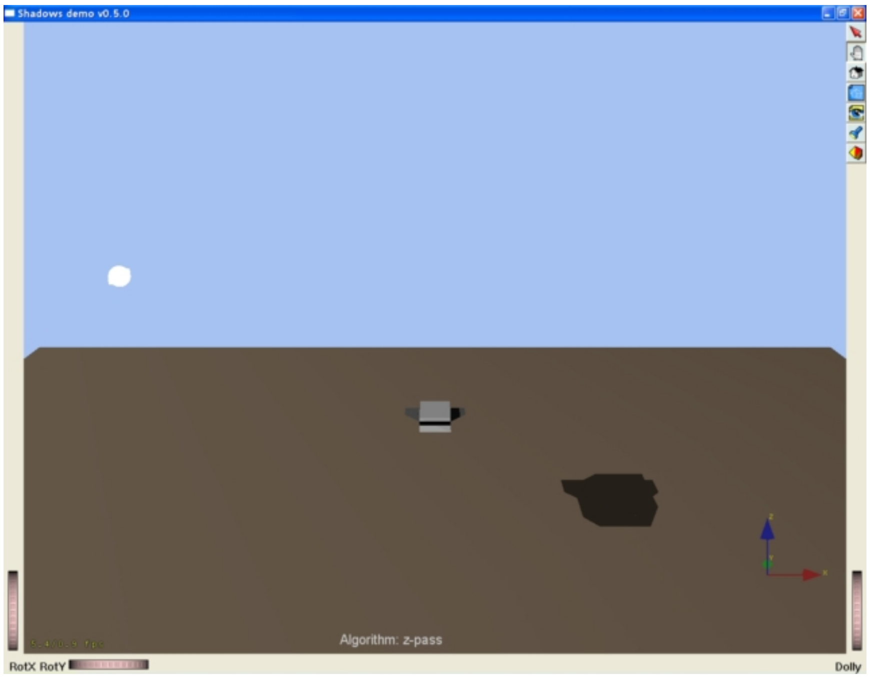

Figure 6.1: Default scene after the start of 'shadows_demo' application.

Key	Action
F1	toggle shadows on / off
F2	toggle z-fail / z-pass algorithm
F3	toggle enable / disable sensors of objects
{/}	increase/decrease light1 intensity
F5/F6	double / half size of object
F7/F8	object's zoom ++ / --
1-4	add / remove new point light
5	add / remove new directional light
6	add / remove new spot light
F9	toggle light1 on / off
F11/F12	increase / decrease ambient light intensity
ARROWS	move object1 along x and y axes
PAGE UP/DOWN	move object1 along z axe
key O	add / remove the second object
key V	export shadow volumes into ShadowVolumes.iv
key X	export scene SceneRoot.iv

Table 6.1: Control keys in action mode of 'shadows_demo'.

Chapter 7

Results

7.1 Speed of the Shadow Engine

The ‘shadows_demo’ application was used to measure speed of shadow computing and rendering. For this purpose, there is a fps (frames per second) counter in the bottom left corner. There are two values, the second one is the frame rate of the whole application, which is important. Its values varies sometimes a lot, therefore the highest measured value was selected.

All tests were realized at this PC configuration: AMD Athlon 1800+, 512 MB RAM, GeForce 4 MX 440, 64 MB, AGP 4x. Default Open Inventor settings was used for all parameters, application window was maximised, and all models were without textures.

Shadow volumes rendering

The first measurement shows, how the speed of shadow volumes rendering is getting slower with the number of lights in the scene and the number of shadow casting object triangles. The test was taken for both depth-pass and depth-fail algorithms when the camera view was moving (shadows are not recomputed in this case).

The rendering speed goes down with the complexity of the object and the number of lights, as shows the table 7.1.

			1 light		2 lights		3 lights	
Model	Triangle cnt.	no shadows	z-pass	z-fail	z-pass	z-fail	z-pass	z-fail
simple_ship	24	56	52.0	52.0	32.0	30.0	30.0	30.0
tank	340	54	30.0	30.0	29.0	29.5	20.0	20.0
fighter	578	53	30.0	29.5	20.0	17.4	15.6	12.7
bomber	912	52	29.4	20.1	14.9	14.1	11.1	10.2
vez	1808	52	16.3	16.0	11.2	9.0	7.3	6.2
stul	2500	59	13.0	12.5	8.7	6.3	5.4	4.3
skyscrpr	3692	45	4.4	5.1	3.2	2.8	1.9	1.9
myobjects	5323	59	8.0	6.2	4.3	3.1	2.8	2.1
miazza	11232	30	3.7	3.0	1.8	1.5	1.2	0.9

Table 7.1: The speed of shadow volumes rendering for different numbers of point lights in the scene.

If there is *n* lights in the scene, the whole scene is rendered in *n + 1* passes, and for every light, its shadow volumes must be rendered twice. For example, if the model consists of 1000 triangles and every one is light-facing (i.e. casts shadow), the shadow volume is of 1000 * (3 * 2) triangles, if depth-pass algorithm is used. If depth-fail is used, two more triangles must be added to each shadow volume, i.e. 1000 * (3 * 2 + 2) triangles.

Since shadow volumes of each model are rendered twice, it is a high amount of polygons to be rendered to create one frame with shadows. Therefore the speed is depending on the number of polygons of the model so much.

Comparison of depth-pass and depth-fail algorithms

The next measurement in the table 7.1 shows the comparison of depth-pass and depth-fail algorithms for different models. The measurement was taken when a point light was moving in the scene. The depth-fail is slower than the depth-pass because of its higher computation time and two more shadow volume triangles, but its visual results are robust. The difference is seen in the graph in the figure 7.1.

Directional light optimalization

To speed up shadows recomputing, a little optimalization for directional light was implemented (shadows are not recomputed, they are only translated). The table 7.1 and the figure 7.2 shows results of the directional light optimalization against a point light for depth-fail algorithms. It introduces an improvement in the speed, but for high polygons counts it drops down as well.

The measurement was taken when the object was moving in the scene by arrow keys.

7.2 Visual results

The *Shadow volumes method* ensures geometrical correctness of the generated shadows and self-shadowing. But the two used algorithms provides different results.

The depth-pass algorithm produces incorrect shadows when the camera origin is inside of any of the shadow volumes (because its shadow volumes are not capped and they are clipped by a near plane), as shows the figure 7.3.

The figure 7.4 shows shadows of the same scene and view generated by the depth-fail algorithm. Shadows are correct in this case.

When shadows are rendered, sometimes there are visible some small ‘dots’ or little artifacts in them. They are probably caused by rendering of some model edges in each pass of the shadow volumes rendering. This problem can be fixed by using suitable OpenGL extension. Problem areas are shown in the figure 7.5.

Model	triangles cnt	depth-pass	depth-fail
simple_ship	24	58.2	58.2
tank	340	31.4	20.5
fighter	578	21.2	12.2
bomber	912	15.6	7.8
vez	1808	9.6	4.3
stul	2500	6.6	1.1
skyscrpr	3692	3.8	2.1
myobjects	5323	3.0	1.3
miazza	11232	0.5	0.4

Table 7.2: Comparison of depth-pass and depth-fail algorithms for different triangles count of the model.

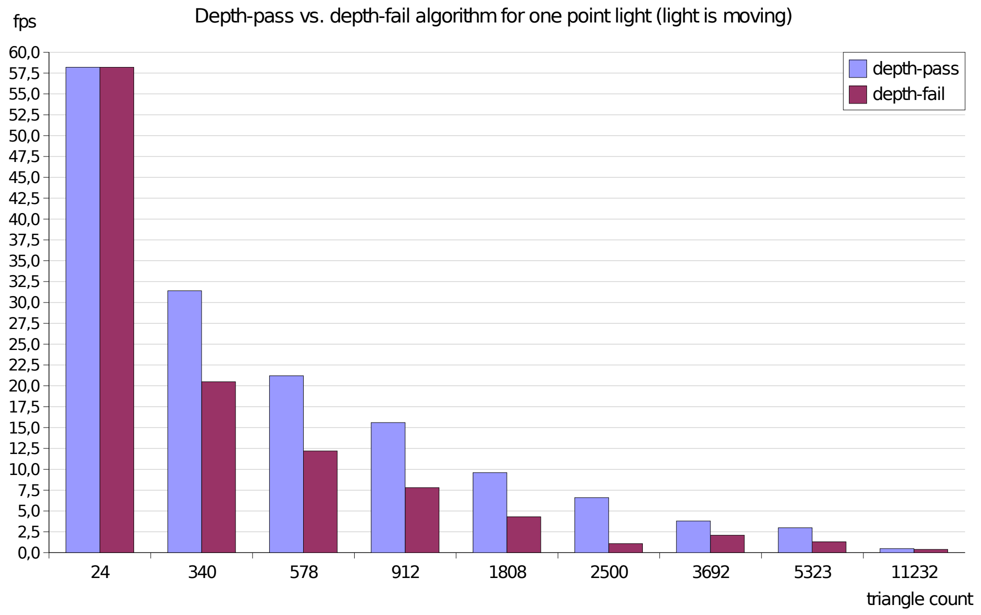

Figure 7.1: Comparison of depth-pass and depth-fail algorithms.

Model	triangles cnt	point	directional
simple_ship	24	31.4	31.5
tank	340	30.8	31.5
fighter	578	22.4	30.5
bomber	912	13.3	30.5
vez	1808	8.5	21.4
stul	2500	4.2	20.8
skyscrpr	3692	1.2	10.3
myobjects	5323	0.9	5.8
miazza	11232	0.2	5.8

Table 7.3: Directional light optimalization against point light.

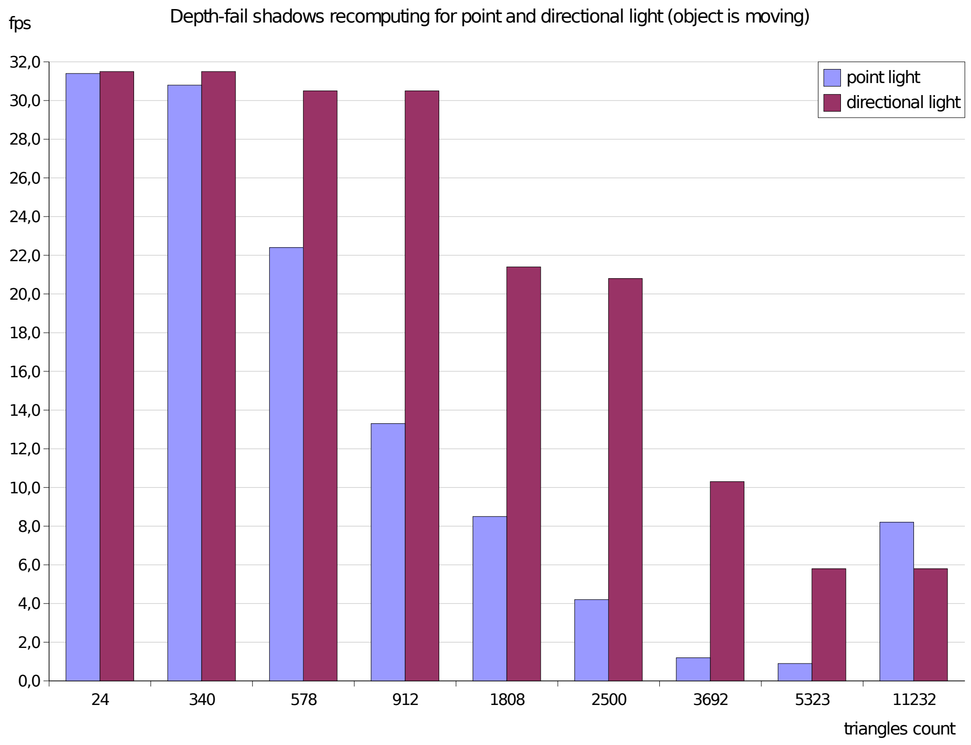

Figure 7.2: Directional light optimalization.

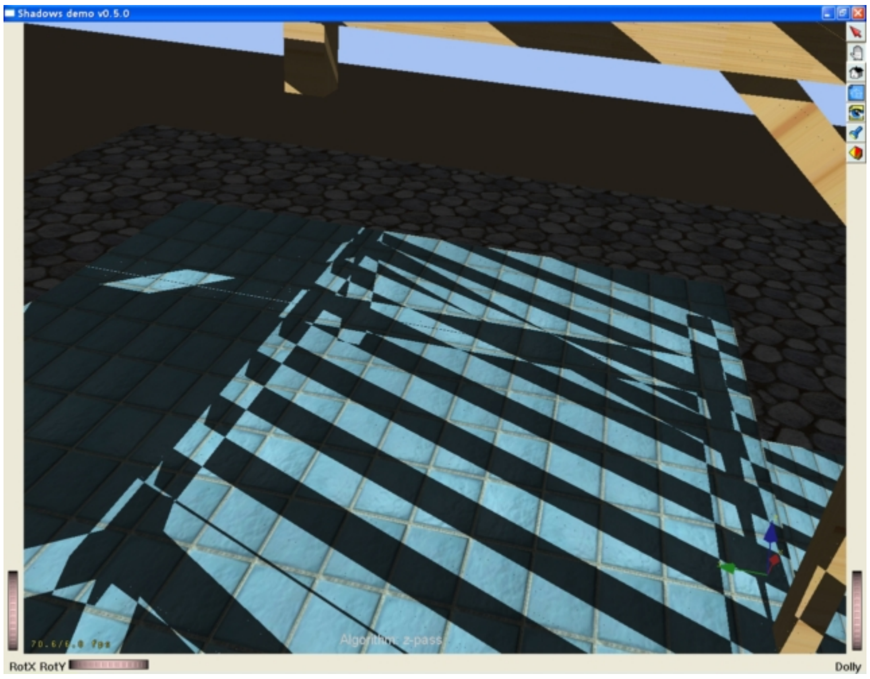

Figure 7.3: Incorrect shadows determination by depth-pass algorithm.

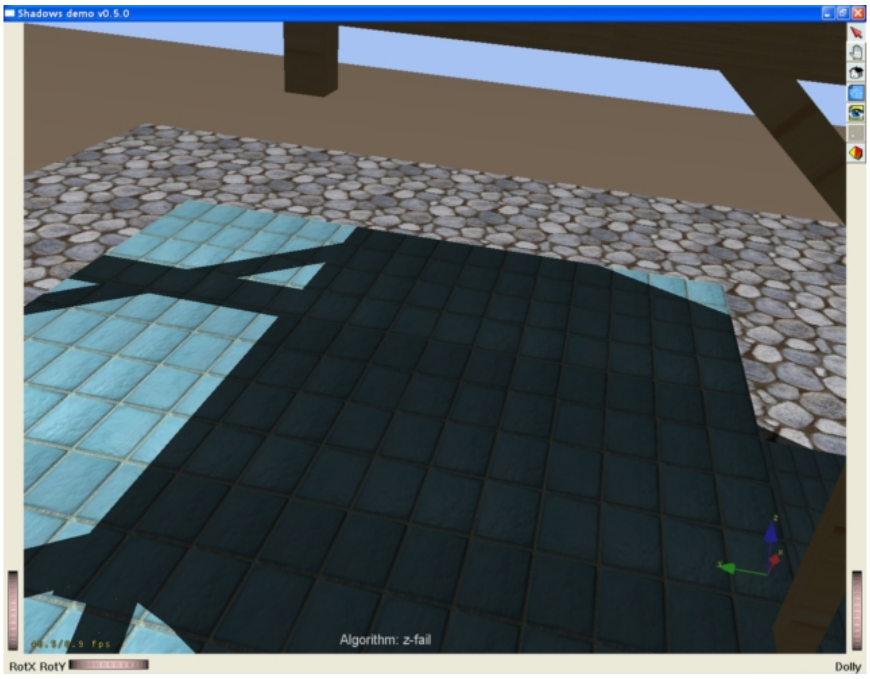

Figure 7.4: Correct result of the depth-fail algorithm for the same view.

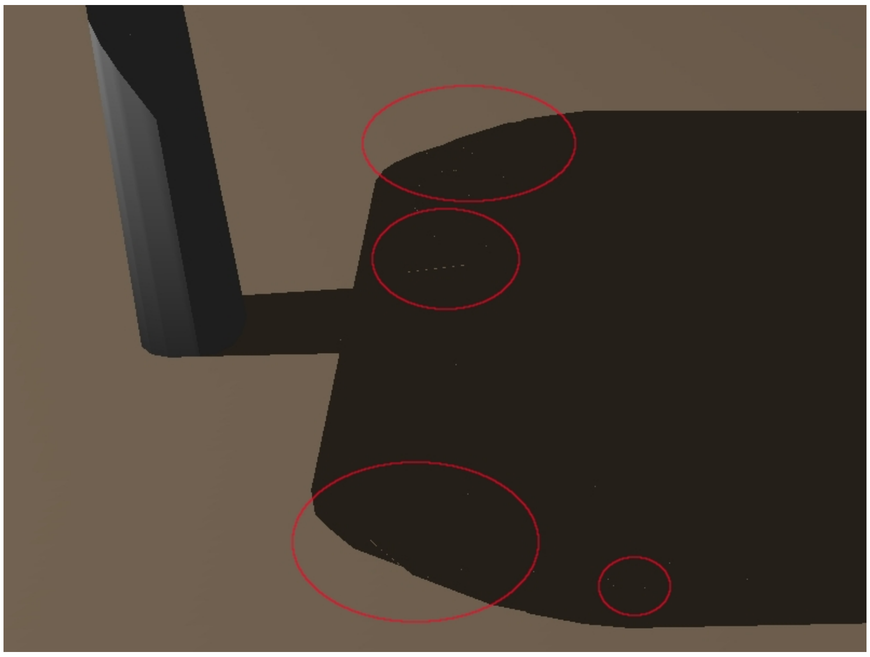
Figure 7.5: Notable artifacts in generated shadows. Problem areas are highlighted.

Chapter 8

Conclusion

8.1 Project analysis

The Shadow Engine project, in my honest opinion, fulfils its expectations and gives a possibility of shadows generating to Open Inventor programmers. Moreover, the usage of the Shadow Engine is easy and platform independent.

According to obtained results I can say that the visual results of shadows are very good, shadows are, as expected, hard-shadows.

Incorrect shadows determination of depth-pass algorithm in certain situations is not a problem, but it is a known property of this algorithm. Nevertheless, this algorithm can be used in applications, where the camera cannot be placed in any shadow volume (e.g. looking down on the scene).

Depth-fail algorithm provides robust shadows in any situations.

The speed of shadow computing and rendering very depends on the number of polygons of shadow casting objects. For objects, that consist of 500 triangles and less, the shadow generation is real-time. For detailed models it slows down with the increasing number of polygons.

The speed of Shadow Engine is probably the highest possible speed that can be achieved by mixing of Open Inventor and OpenGL code at this level. Realtime shadow algorithms are, in general, implemented at lower levels, very often directly in graphic processing units in vertex shaders, to be really real-time.

Thanks to the improvement by the directional light optimalization, the Shadow Engine can be used in small games and CAD applications, where there are mainly directional lights used and objects are moving.

8.2 Next work

Future work should try to improve the speed of the Shadow Engine to work in real-time, even for large scenes. This can be achieved in a number of ways.

At first, the silhouette determination can be implemented. It lowers the number of light-facing polygons and thus the amount of shadow volumes triangles, therefore it should provide better results both for shadow volumes computation and rendering. It is usually done in GPU. But implementation of these algorithms is not trivial.

The speed different between the depth-pass and depth-fail algorithms is clear from the results. In many cases, it is not necessary to use robust, but slower, depth-fail algorithm.

It is possible to switch between the algorithms according to the reciprocal positions of objects and lights in the scene, when the application is running.

A different algorithm can be used to render shadows of each shadow casting object. The Shadow Engine is prepared for this improvement now (the possible value 'autoset' of the Shadow Manager method 'setMethod()').

Many other optimalization techniques can be used as well, for example the level of detail can reduce the number of triangles for shadow volumes computation, as well as scene clipping.

Next work should also remove shadow artifacts, and do some tuning of the complete engine. It would be nice to transform used algorithms to cast soft-shadows as well. It is probably worth trying to implement another shadow method, e.g. *shadow maps*, in the Shadow Engine.

Bibliography

[1] Coin3D, SGI Open Inventor 2.1 fully compatible 3D high-level graphic toolkit.
`http://coin3d.org`.

[2] Lexoview homepage at SourceForge.net.
`http://sourceforge.net/projects/lexoview`.

[3] Open Inventor projects at FIT, Brno University of Technology.
`http://merlin.fit.vutbr.cz/wiki/index.php?title=Inventor_Projects_2006`, 2006.

[4] David Ambrož. Shadows techniques.
`http://www.shadowstechniques.com/intro_cz.html`.

[5] Frank C. Crow. Shadow Algorithm for Computer Graphics. Proceedings of SIGGRAPH '77, vol.11, no. 3, pp. 242-248.

[6] Cass Everitt. Shadow Mapping. `http://developer.nvidia.com/attach/6393`.

[7] Cass Everitt and Mark J. Kilgard. Optimized Stencil Shadow Volumes.
`http://developer.nvidia.com`, 2003. GDC 2003 presentation.

[8] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of Real-Time Soft Shadows Algorithms.
`http://artis.inrialpes.fr/Publications/2003/HLHS03a/`. Computer Graphics Forum, Volume 22, Number 4, page 753–774 - Dec. 2003.

[9] Tim Heidmann. Real Shadows, Real Time. 18:28–31, 1991.

[10] Hun Yen Kwoon. The Theory of Stencil Shadow Volumes.
`http://www.gamedev.net/reference/articles/article1873.asp`.

[11] Eric Lengyel. The Mechanics of Robust Stencil Shadows.
`http://www.gamasutra.com/features/20021011/lengyel_01.htm`, 2002.

[12] Ing. Jan Pečiva. Open Inventor tutorial at Root.cz.
`http://www.root.cz/clanky/open-inventor/`.

[13] Lund University. Soft shadow research at Lund University.
`http://graphics.cs.lth.se/research/shadows/`.

[14] Josie Wernecke. *The Inventor Mentor*. Addison-Wesley Professional, 1994.
ISBN 0201624958.

[15] Jiří Žára, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika*. Computer Press, 2004. ISBN 80-251-0454-0.

Figures 3.1, 3.2, 3.4 were taken from *http://www.shadowstechniques.com* with kind permission of David Ambrož.

Figures 3.3, 3.5, 3.6 were taken from *http://www.gamedev.net/reference/articles/article1873.asp* with kind permission of Hun Yen Kwoon.

Chapter 9

Image appendix

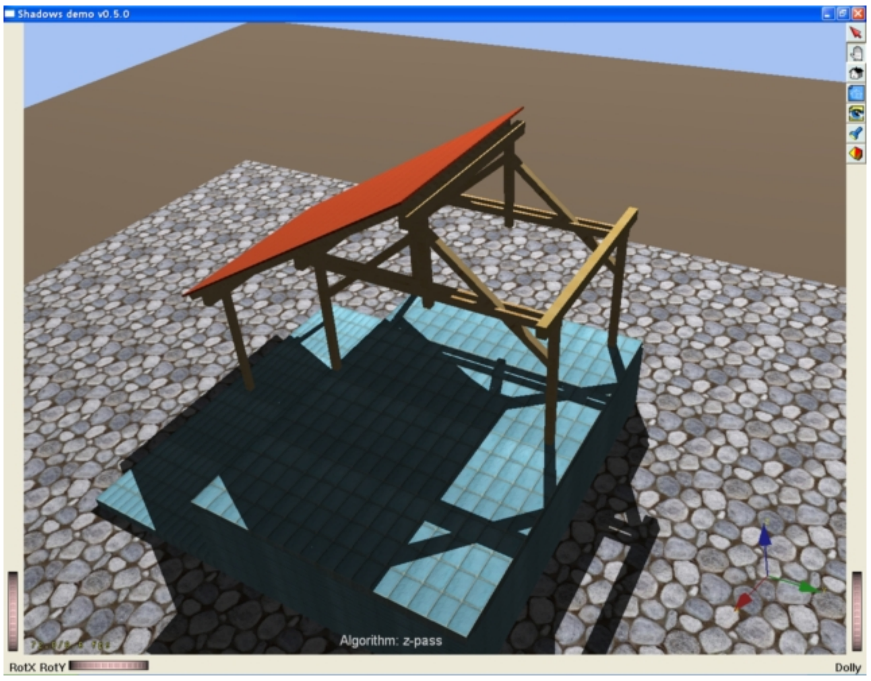

Figure 9.1: Shadows of a wooden construction.

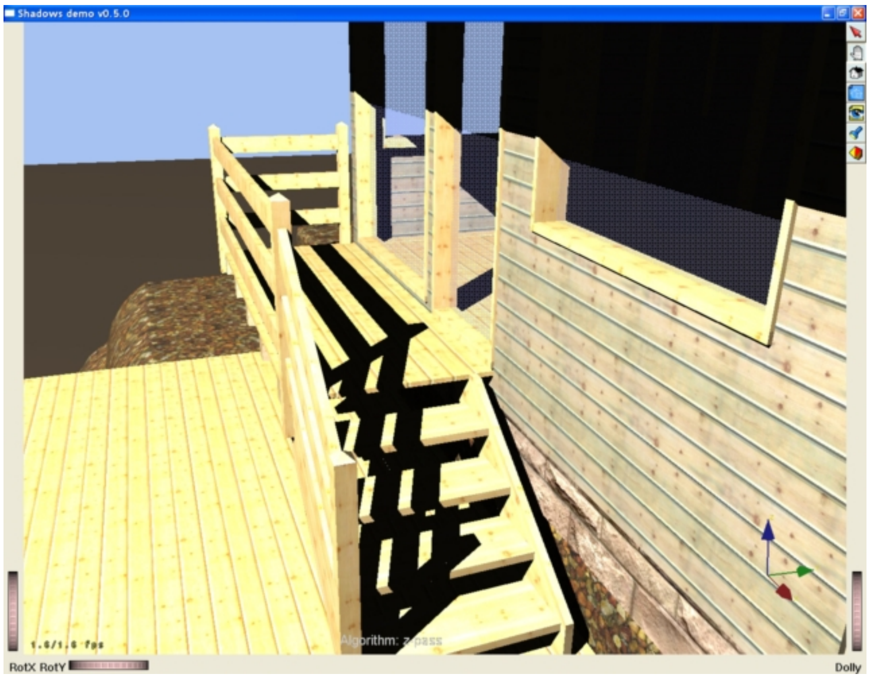

Figure 9.2: En example of generated shadows can copy the surface of objects. Note also the shadow visible trough the transparent window.

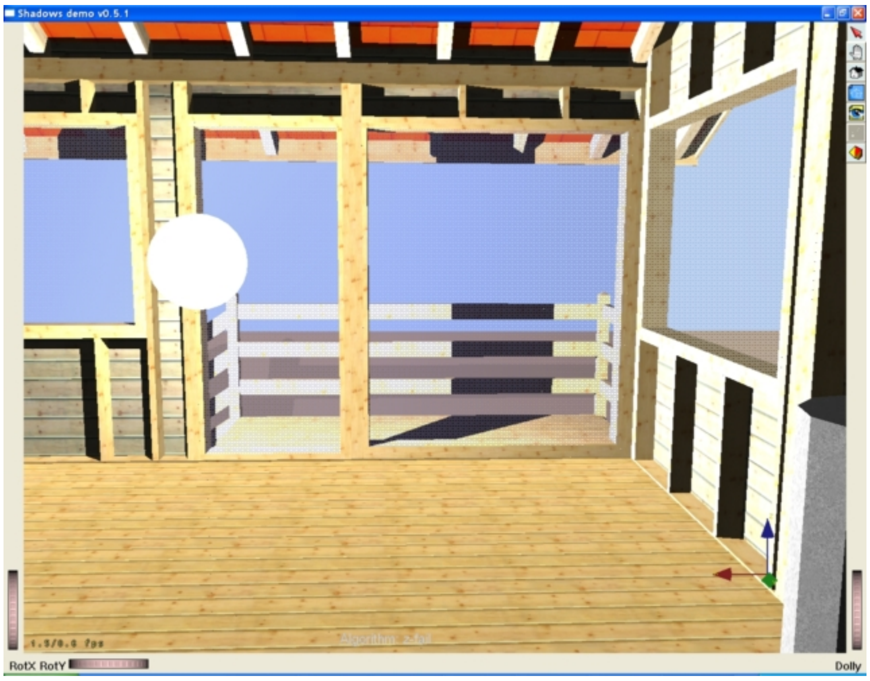

Figure 9.3: Shadows from the inside of the building. The shadow of the pole is visible through the window. The white ball is a point light source.

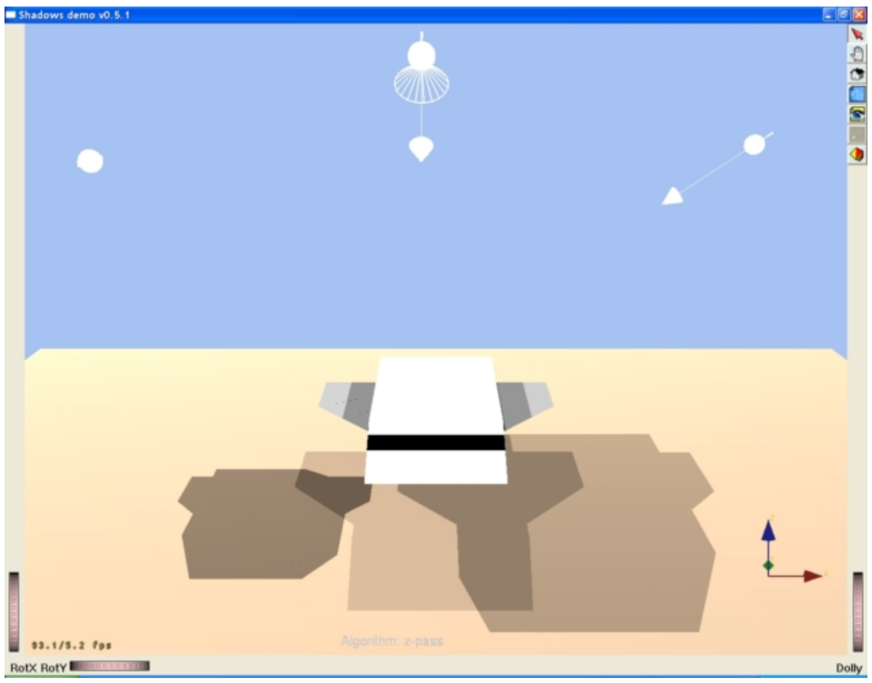

Figure 9.4: Example of shadows of all light source types of Open Inventor (point light, spot light and directional light).

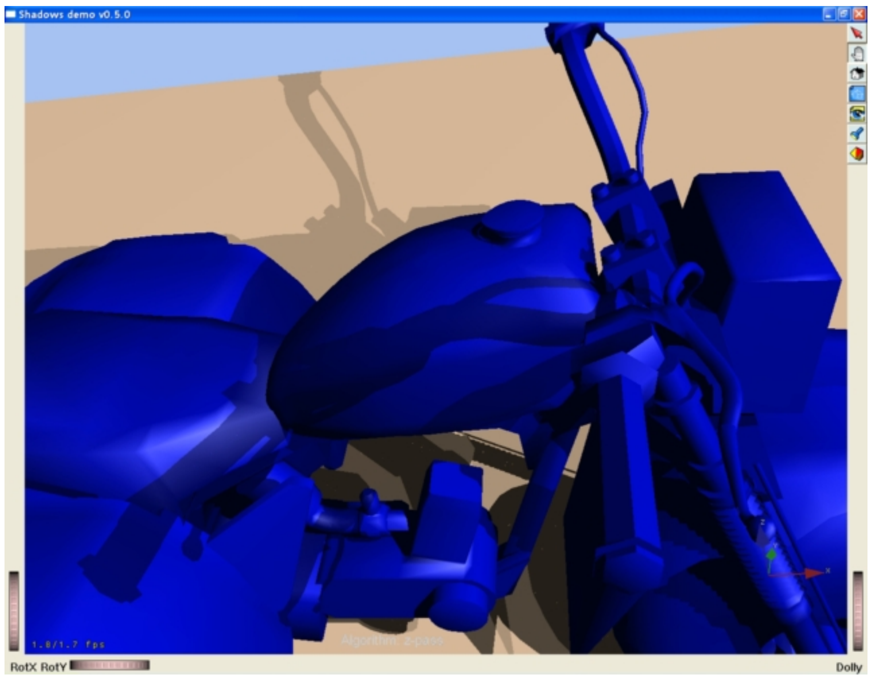

Figure 9.5: Self-shadowing on a motorbike.

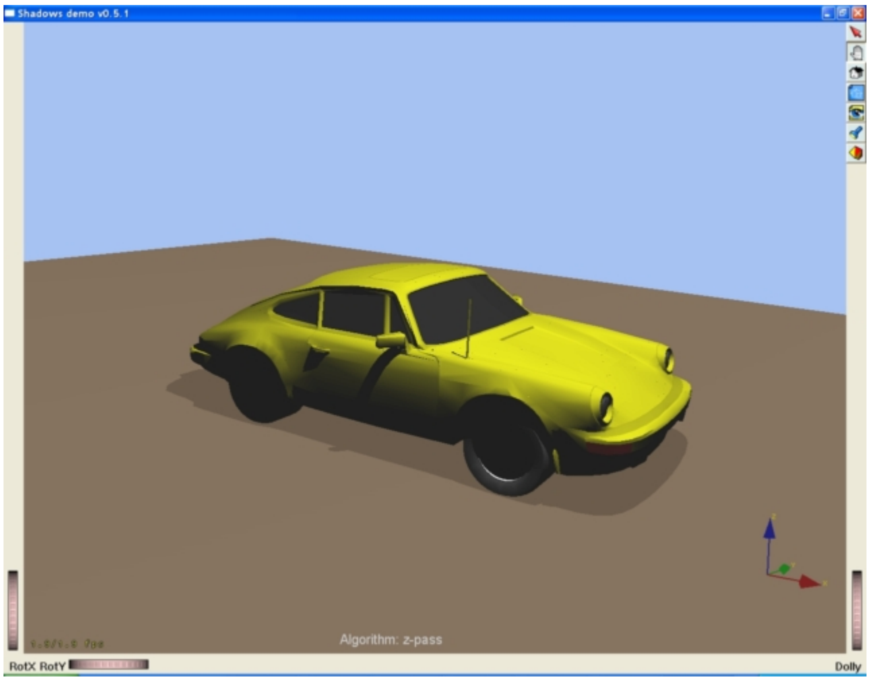

Figure 9.6: High-detail porsche model shadows.

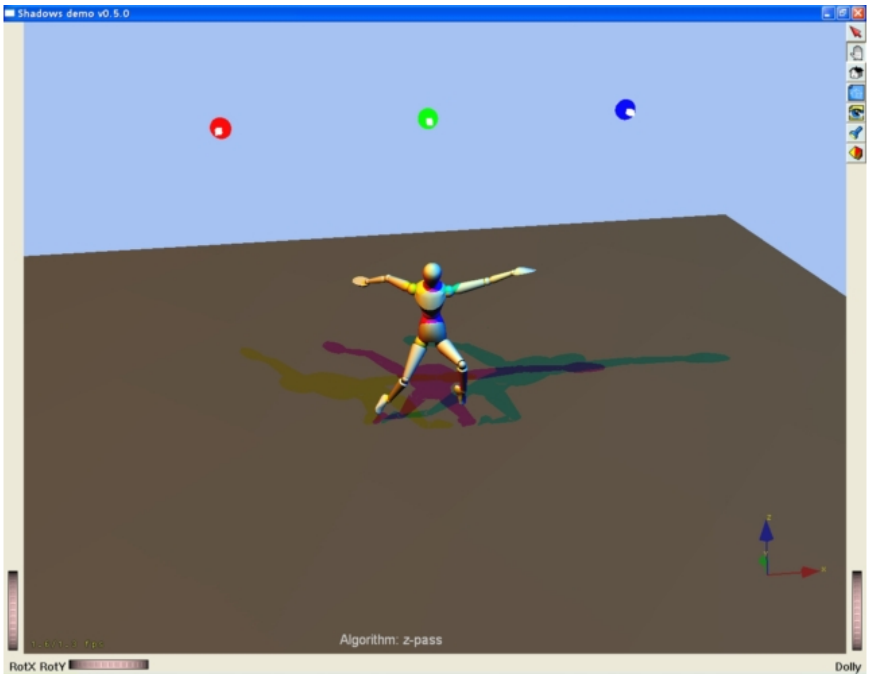

Figure 9.7: Correctly coloured shadows of three light sources.

Chapter 10

The Shadow Engine minimal application example

```
/* ---------------------------------------------------------------------------
 * Authors:      Tomáš Burian (tburian _AT centrum.cz)
 *
 * THIS SOFTWARE IS NOT COPYRIGHTED
 *
 * This source code is offered for use in the public domain.
 * You may use, modify or distribute it freely.
 *
 * This source code is distributed in the hope that it will be useful but
 * WITHOUT ANY WARRANTY.  ALL WARRANTIES, EXPRESS OR IMPLIED ARE HEREBY
 * DISCLAIMED.  This includes but is not limited to warranties of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 * If you find the source code useful, authors will kindly welcome
 * if you give them credit and keep their names with their source code,
 * but do not feel to be forced to do so.
 * ---------------------------------------------------------------------------
 */


//----------------------------------------------------------------------------
// INCLUDEs
//----------------------------------------------------------------------------

#define APPNAME ``SE mini''
#define VERSION ``v0.3''

#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/manips/SoPointLightManip.h>
#include <Inventor/Win/viewers/SoWinExaminerViewer.h>
#include <Inventor/Win/SoWin.h>
```

```
#include <Inventor/SoDB.h>
//
#include ``ShadowEngine/ShadowManager.h''

//-------------------------------------------------------------------------------
// IMPLEMENTATION
//-------------------------------------------------------------------------------

// Linker: Link with main (not WinMain)
// and use a console
#pragma comment(linker, ``/entry:\''mainCRTStartup\`''')
#pragma comment(linker, ``/subsystem:\''console\`''')

char title[50] = ``'';


int main(int argc, char **argv)
{
  SoDB::init();

  // Show FPS
  putenv(``IV_SEPARATOR_MAX_CACHES=0'');
  putenv(``COIN_SHOW_FPS_COUNTER=1'');

  // Initialisation
  HWND window = SoWin::init(argv[0]);
  if (window == NULL) exit(1);

  // This is the highest scene root (with shadow scene) that will
  // be shown in viewer; camera is added to it ass well
  SoSeparator * topRoot = new SoSeparator();
  topRoot->ref();

  // Root of the user's scene
  // create your scene under this root node
  SoSeparator * root = new SoSeparator();

  // Camera
  SoPerspectiveCamera * camera = new SoPerspectiveCamera;
  camera->position = SbVec3f(0.f, 20.f, 50.f);
  camera->pointAt(SbVec3f(0, 0, 0));

  // Light 1
  SoPointLightManip * light1 = new SoPointLightManip;
  light1->location.setValue(30.0f, 70.0f, 40.0f);
  light1->intensity.setValue(0.5f);
  light1->color.setValue(SbColor(0.9f,0.9f,0.9f));
  light1->on = true;
  root->addChild(light1);

  // First, create a parent node for a new object
  SoSeparator * objectParent = new SoSeparator;

  // Load object
  const char * filename = ‘‘stonehenge.iv’’;
  SoInput in;
  if (!in.openFile(filename)) {
    printf(‘‘Can not open input file ’%s’.\n’’, filename);
    exit(-1);
  }
  SoSeparator *object = SoDB::readAll(&in);
  if (object == NULL) {
    printf(‘‘Can not read input file ’%s’.\n’’, filename);
    exit(-1);
  }

  objectParent->addChild(object);

  root->addChild(objectParent);


// Shadow Manager (SM) /////////////////////////////////////////////////////

  // Create Shadow manager based on user’s scene root and environment node.
  CShadowManager * shadowManager = new CShadowManager(root, NULL);

  // register lights and objects in SM
  shadowManager->addLight(light1);
  shadowManager->addObject(object);

  // select shadow computation algorithm
  shadowManager->setMethod(zfail);

  // create shadow scene
  topRoot->addChild(camera);
  topRoot->addChild(shadowManager->getShadowSceneRoot());

// Shadow Manager end //////////////////////////////////////////////////////

  // Create viewer
  SoWinExaminerViewer *viewer = new SoWinExaminerViewer(window);

  // Shadow Manager: TURN ON THE STENCIL BUFFER SUPPORT !!!
  // this must be done to see any shadows
  viewer->setStencilBuffer(TRUE);
  // because of render area, viewers do that automaticaly
  camera->viewAll(topRoot, viewer->getViewportRegion());


  viewer->setSceneGraph(topRoot);
  sprintf(title,‘‘%s %s’’, APPNAME, VERSION);
  viewer->setTitle(title);

  viewer->show();

  // Show window and start render loop
  SoWin::show(window);
  SoWin::mainLoop();

  // Free memory
  delete viewer;
  topRoot->unref();

  return 0;
}
```