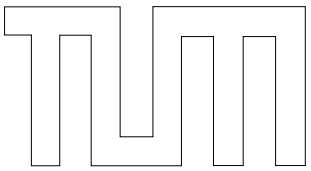
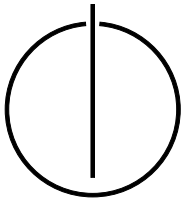


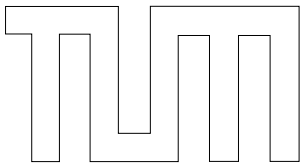
Fakultät für Informatik
der Technischen Universität München

Diplomarbeit in Informatik

Visualization of large-scale 3D city models with detailed shadows

Matthias Wagner





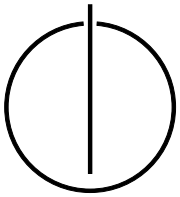
Fakultät für Informatik
der Technischen Universität München

Diplomarbeit in Informatik

Visualization of large-scale 3D city models with detailed shadows

Visualisierung großflächiger 3D Stadtmodelle mit detaillierten Schatten

Author: Matthias Wagner
Supervisor: Prof. Dr. Rüdiger Westermann
Advisor: M.Sc. Stefan Hertel
Submission Date: June 16th, 2008



I assure the single handed composition of this diploma thesis only supported by declared resources.

June 16th, 2008 _____

Matthias Wagner

Abstract:

With increasing processing power and bandwidth, large-scale environments are becoming more common in computer applications. Users are able to browse maps of the whole world or walk through 3D models of cities using their standard web browser. However, this is not limited to the visualization industry. Entertainment products like games are shifting towards larger environments, too. Some games are even simulating whole worlds. However, consumers still expect high-quality display, which can be harder to achieve for large scenes.

The thesis provides a short introduction to CityGML, which has been chosen as data source. It then shows how to efficiently render large cities by exploiting spatial coherency and occlusion information. Spatial subdivision is carefully used to achieve this. The thesis also covers how to do this for time dependent scenes, e.g. cities developing over centuries. Another important requirement for efficient rendering is to avoid state changes, which can be reduced by batching draw calls based on material.

As sunlight shadows are necessary to display cities realistically, they are considered an essential part of the thesis. Several methods for rendering dynamic shadows are presented. Focus is put on shadow maps, which provide a good trade-off between speed and quality. Several methods for improving shadow map quality are proposed, some increasing resolution close to the camera and some hiding aliasing by smoothing.

Although shadow maps can be improved significantly, aliasing artifacts usually cannot be avoided. The thesis therefore presents a hybrid approach of shadow maps and ray tracing. Ray tracing shadows is quite expensive but offers high quality, while shadow maps are comparatively cheap but have low quality. Combining both techniques results in high-quality shadows with much less artifacts at competitive speed. In future, this approach might even be interesting for real-time applications like games.

Next, an overview of the design of the developed application is given and performance of the different proposed techniques is compared. Finally, the thesis is concluded by presenting possible extensions to the application along with a valuation of future possibilities.

Contents

Chapter 1

Introduction

Many people have enjoyed browsing maps of landscapes and cities using a web browser. Internet portals providing those services became very popular, as their maps can be used for planning trips, calculating distances or finding hotels or interesting locations. Recently there have been several approaches to extent these services to provide fully three dimensional models of cities to the user.

In this thesis a viewer for three-dimensional models of cities is presented. For the course of the thesis the viewer is simply called *City Viewer*. As focus is put on modern visualization techniques, basic knowledge about computer graphics and rendering *application programming interfaces* (APIs) like DirectX and OpenGL is assumed. The reader is also expected to know the *Extensible Markup Language (XML)*.

1.1 City models

This section takes a look at common standards that are used for storing city data. While there are many formats available to store geometry, only those that are well suited for city data are considered.

1.1.1 Google Earth

Google Earth is a very popular viewer for Google's own *Keyhole Markup Language (KML)*, which is based on XML. It doesn't only support specification of geometry but also other parameters like view angles or distances. Linking to data on internet servers is possible, too.

KML recently became an *Open Geospatial Consortium (OGC)* standard. However, *City Viewer* currently does not support KML in favor of *CityGML*, which is also an OGC standard.

1.1.2 CityGML

CityGML is a standard of the *Open Geospatial Consortium*. In contrast to other standards, it focuses on much more than just visualization. CityGML includes semantic information like traffic models or vegetation. Kolbe [Kol07a] reasons that there is currently a paradigm shift in spatial modeling away from graphic models to well-defined objects with properties, structures and interrelationships. He states that 3D city and landscape models are a product family on their own with specific applications. They fully representate city topography and structures *as observed* and usually not *as planned*. Additionally, they provide approximately homogenous data quality for the whole model regarding geometry, topology, semantics and appearance.

1.1.3 Use cases

City models may be used for many purposes. They can be used for planning sighting trips or finding appropriate places to sleep. Or they can serve completely different and probably unobvious purposes, some of which will be shortly presented here.

Cartography

Usually the user doesn't only want to look at the outer appearance of objects, but he is also interested in retrieving semantic information about them. For example, it may be interesting to know if there are hotels, filling stations or parks along a route. A viewer tool can then strip away information that is not necessary for the current task, providing a clear display to the user.

Navigation

If the model provides good semantic information, the user may request tours that highlight certain aspects like famous sightings or interesting historic locations. This tour can then be created (semi-)automatically by a tool.

Simulation

Authorities may use the model to simulate natural catastrophes like floods to take precautions. Of course, this may also be interesting to possible buyers of estates. This leads to another related issue: the user can check lines of sight from a location to another before making his decision. A particularly interesting application in that regard is noise simulation. The state of North-Rhine Westphalia in Germany has extended CityGML to support noise mapping to fulfill the requirements of the *European Commission Environmental Noise Directive* [PC02].

1.2 Viewer requirements

In the course of this thesis a viewer for CityGML files shall be developed. Primary focus is taken on the actual display of 3D cities due to time constraints. Therefore, applications like simulation or navigation are currently not supported. However, the design of the viewer application should allow for possible extensions. This section will explain the most important requirements imposed on the development.

1.2.1 Functional requirements

CityGML support

The system must be able to load CityGML files and display them in 3D. It should support all basic parts of CityGML including appearance themes.

Picking

The user can pick specific surfaces, building parts, buildings and other city objects. The viewer then displays information about the picked part in an intuitive way.

Display

Additionally to CityGML appearance theme support, the viewer should provide basic texturing and coloring based on semantic information.

Shadows

Sun shadows are considered an important part of the viewer, therefore good quality shadows must be provided even for large cities (while staying interactive).

1.2.2 Performance requirements

Display of the city shall be accelerated and improved by using modern techniques. This involves usage of occlusion culling and state-of-the-art shadow generation. The target frame-rate is about 25 FPS with medium quality settings and medium sized cities on current standard hardware (P4 Dual 3GhZ, NVIDIA Geforce 8800 GTS, 2GB RAM).

1.3 Platform

In order to keep the actual viewer application simple while still having access to modern technologies like DirectX 10, quite strong platform requirements have been set. However, the source code must be organized modularly, so that porting the viewer to other platforms is eased.

1.3.1 Operating environment

The viewer will use DirectX 10 for displaying the city to the user. This implies Microsoft Windows Vista as target platform, along with a DirectX 10 compatible graphics card. However, modules of the application that don't have to do with actual display will be developed using platform independent technologies.

Chapter 2

CityGML

According to Kolbe [Kol07b] *CityGML* is a common information model for representing 3D urban city objects. It specifies the classes and relations for topographic objects in cities. Those classes do not only contain geometry and appearance information but also semantic and topological properties, therefore allowing for a widespread use of the *CityGML* model. *CityGML* files, an extension of *GML* files which are stored using *XML*, are used as data source for *City Viewer*. *CityGML* stores a hierarchy of buildings and city objects. The objects themselves store their associated geometry data using *GML*, while their appearance, including texture coordinates, is specified using *CityGML's* appearance model. The version of *CityGML* used for this thesis is *0.4.0*, which is an application schema for *GML 3.1.1*.

2.1 Introduction

While many formats are available to store geometry of a city model, there has been a lack of formats for storing semantic and topological data of those cities. *CityGML* tries to satisfy the need of applications other than visualization for this data. It is developed by members of the *Special Interest Group 3D* of the initiative *Geodate Infrastructure North-Rhine Westphalia*. As this thesis is mainly about visualization of *CityGML* models, this introduction focuses on the visualization specifics of *CityGML*.

2.2 CityGML

CityGML (and *GML*) defines classes and their relations to each other, therefore the *Unified Modelling Language (UML)* is used to specify class models. *UML* class models

also lend themselves nicely for implementation. The specification of *CityGML* [GKC07] and *GML* [CDL+07] includes both XML schemata and UML models.

2.2.1 Themes

CityGML has a widespread theme model, reaching from terrain reliefs to detailed building models. Figure 2.1 gives an overview of some parts of the CityGML model. The CityGML specification [GKC07] includes more details about all models included in CityGML.

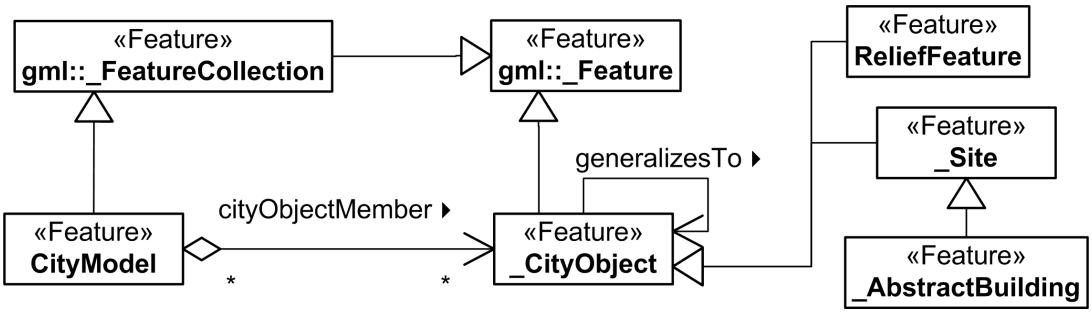


Figure 2.1: Excerpt of CityGML themes

The abstract base class of almost all CityGML classes is *_CityObject*. It holds creation and termination dates of the model along with an arbitrary amount of string, integer, double, date and URI attributes. Each of those attributes is a pair of name and value and therefore allows for easy and minimal invasive extension of CityGML objects. In the context of this thesis, the string attributes "constructionDate" and "destructionDate" are interpreted in order to include time information.

The following sections will give a slightly more detailed view on chosen models of CityGML. The specification [GKC07] includes full details on each model.

2.2.2 Level of detail

CityGML provides support for five levels of detail (LOD0 - LOD4). At higher LODs, more data is available. For example, in LOD4 rooms may have furniture objects, while in LOD3 room furniture are not available. While this LOD model is nice for choosing a general LOD when viewing a city, it differs greatly from the LOD models usually used in computer graphics. A common LOD model used in computer graphics contains the same geometry in differing detail. That way, the LOD can gradually be changed depending on distance to the camera. The CityGML LOD model instead

removes whole geometry objects in lower detail levels. As this would clearly be visible while exploring the city, *City Viewer* currently only displays one level of detail at once. If memory issues arise, *City Viewer*'s design supports adding dynamic LOD functionality.

2.2.3 Building model

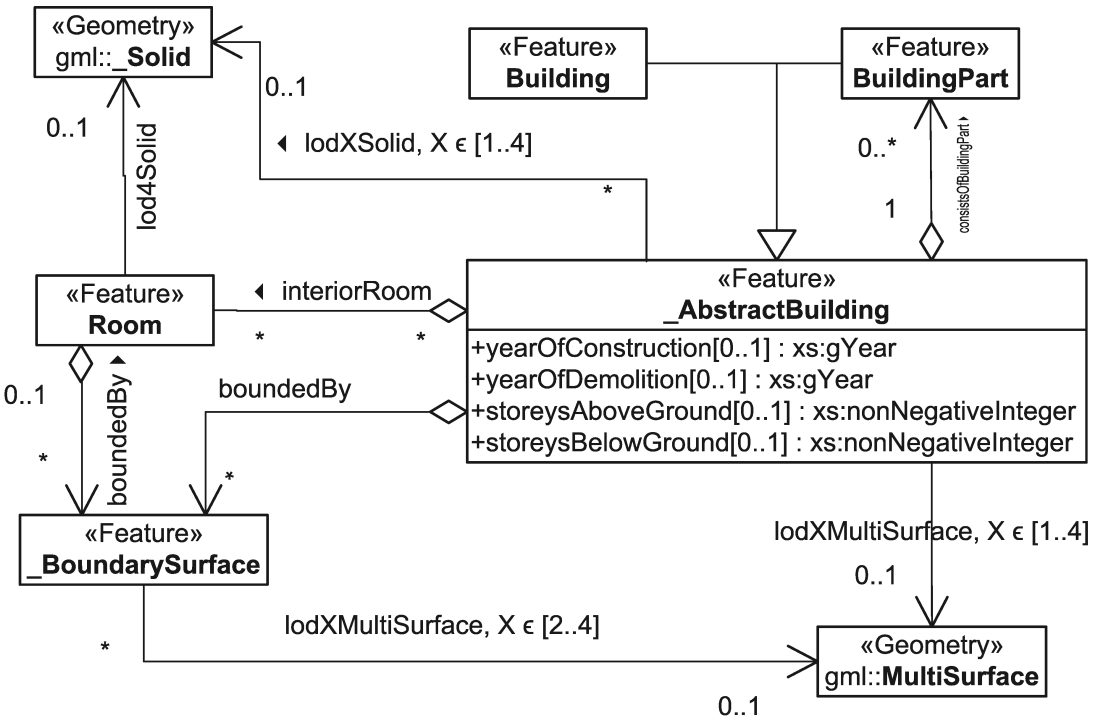


Figure 2.2: Excerpt of CityGML building model

The CityGML building model is probably the most interesting for the development of *City Viewer*. Parts of the building model are shown in figure 2.2. According to the class model, a building may consist of several building parts, which themselves may recursively contain building parts. This allows to model huge buildings which are composed of building parts that have different characteristics, like story count or construction dates. Depending on the level of detail (LOD), only certain parts of the class model apply.

The actual building model is much more detailed. For example, there are several subclasses of *_BoundarySurface*, like *RoofSurface* or *WallSurface*, and each of these surfaces may contain doors and windows. For higher levels of detail buildings and rooms may contain installations and furniture. Additionally, doors and buildings may reference several *Address* objects, which can be shared like most GML objects, meaning more than one door can be connected to an address.

The different *_BoundarySurface* types allow viewing tools to decide on coloring or texturing of objects, if no appearance information is available for the surface. *City Viewer* takes use of this, as most example datasets available were modeled without appearance information.

2.2.4 Appearance model

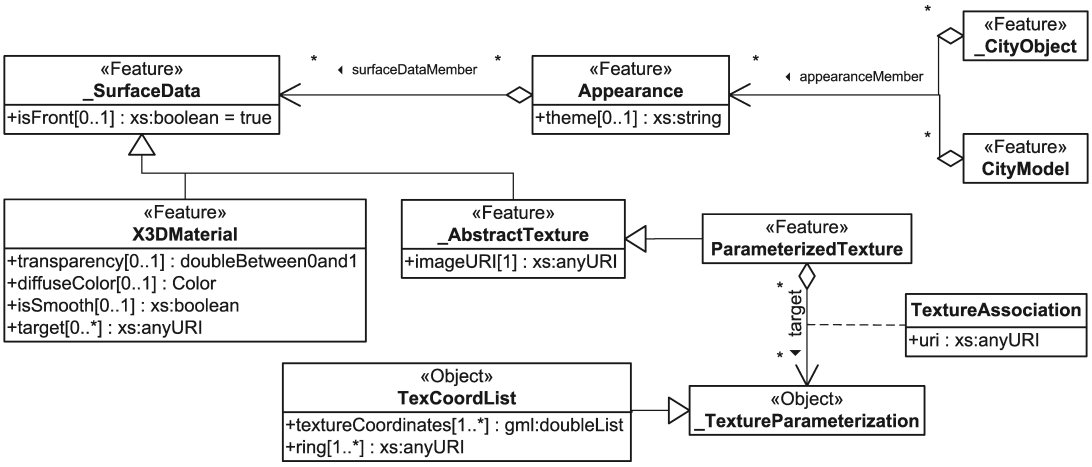


Figure 2.3: Excerpt of CityGML appearance model

CityGML provides an advanced appearance model, including several possible display themes for the same city model. These themes specify the current look of the model. For example, one theme could be "winter" and another one "spring". However, themes are not limited to the different times of the year, but can also be used to display other information like heat.

Applying display themes to surfaces

City objects (and the city model itself) can contain appearances for different themes. Each of those appearances holds several *_SurfaceData* objects, which can be materials or textures. These are applied to the actual geometry surfaces. The exact way of doing this depends on the type of *_SurfaceData* object and is explained below. Generally, they are linked to one or more surfaces. This means that the surface itself does not know anything about the way it should be displayed!

Materials Materials hold general surface information like shininess, transparency, or specular and diffuse color. A material also contains a boolean flag whether surfaces with this material shall be smoothed. Smoothed materials use vertex normals, while

other materials use face normals. The first should be used for curved surfaces, while the latter is important to generate properly shaded walls with visible edges.

Each material links to a list of surfaces it is applied to. Each surface usually should be linked to by at most one material of each theme. If more materials of the same theme link to a surface, it is up to the application to choose which one to use.

Textures There are many ways to specify textures for surfaces in CityGML. However, each texture class is derived from *_AbstractTexture*, which contains generic information like texture image or wrap mode. One texture class is *GeoreferencedTexture*, which uses textures that are aligned to a certain world reference point. The other texture class if *ParameterizedTexture*, the type of texture most commonly used in graphics applications. Parameterized textures use texture coordinates for each vertex of a surface. Thus they are set for each target surface independently. Each surface can either receive texture coordinates by multiplying the world position of the vertices with a worldToTexture matrix, or directly using a *TexCoordList*. Setting texture coordinates directly uses the XML linking feature, similarly to linking to the surface. As is explained in the next section, GML specifies polygons through inner and outer rings. Therefore each vertex of each ring contained in the surface must receive texture coordinates. This is achieved by two lists of the same size, one being the textureCoordinates list and the other being the appropriate ring list[1]. The order of the elements in these lists is therefore important, as each ring must be mapped to a texture coordinates list.

2.3 GML

The *Geography Markup Language (GML)* is a XML grammar used to model, transport and store geographic information [CDL+07]. It provides support for coordinate reference systems, geometry, topology, time, units of measure and storage of geographic information. This section focuses on explaining the geometry model.

2.3.1 Geometry model

The GML geometry model is a hierarchical structure. Geometries may not only be specified using vertices and indices like in most model formats. Instead, the model is

[1] Actually, the textureCoordinates list is twice the size of the ring list, as each texture coordinate consists of two components

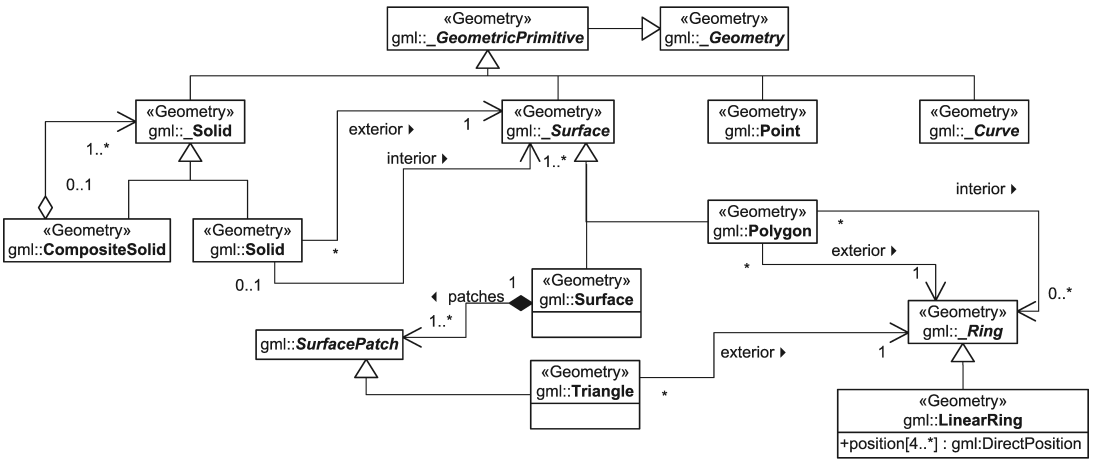


Figure 2.4: Excerpt of CityGML specific GML geometry model

based very much on aggregation and the composite pattern.

Most primitive types are modeled using aggregation. Solids are specified by exactly one exterior surface and optionally several interior surfaces. The surface itself is described by exactly one exterior ring and, depending on the type of surface, optionally several interior rings.

A composite pattern is used for modeling complex solids, surfaces and curves. This allows to reproduce real world primitives more closely, as those usually can be seen as composition, too.

2.3.2 Triangulation

As explained before, geometry in GML usually is not given as triangle lists. To render the data using a GPU, all surfaces must be triangulated. *City Viewer* does this by using the "OpenGL Graphics System Utility Library", which offers polygon tessellation including support for interior contours.

Chapter 3

Efficient data structure

This chapter discusses the way *City Viewer* handles geometry data. The geometry data GML provides is not directly usable for rendering. Even after triangulating the data and uploading it to the graphics card, simply iterating through all CityGML objects along with their attached GML geometries is inefficient. While each GML surface usually just holds a few vertices, there is a very large number of surfaces. This requires many small draw calls, each with possibly different rendering effects, which is prohibitive for achieving interactive frame rates.

Sorting the surfaces by effect type and drawing all surfaces of one effect type at once can be done to dramatically improve rendering speed. However, as *City Viewer* shall handle very large environments with high detail, this can still be extremely slow due to completely ignoring line of sight. By using an intelligent data structure several technologies can be employed, including occlusion or frustum culling, which can improve rendering performance drastically if large parts of the scene are currently not visible.

Therefore, several well known and efficient data structures for storing geometry data are presented in this chapter. The chapter also handles the issue of storing time dependent information. This is important, as *City Viewer* may be used to show the development of a city over several centuries. If the city changes shape rapidly several times, this can result in a high amount of data to be handled.

3.1 Common acceleration structures

This section introduces some common acceleration structures frequently used in computer graphics. Many of these techniques originally are targeted at improving ray

tracing performance. However, these structures usually map well to rasterization algorithms and occlusion culling techniques.

3.1.1 Bounding volumes

A complex object with thousands of triangles can be approximated by much simpler geometric primitives. For example, a whole building containing furnitures, doors, windows and other details can be approximated by a bounding box. Instead of testing each triangle of the building for an intersection or visibility in the view frustum, the bounding volume can be tested first. This is usually much cheaper than testing the real geometry. Like most other viewing tools, *City Viewer* therefore supports this technique using bounding boxes[1], as it provides a significant speedup for negligible implementation and memory effort.

Kay and Kajiya [KK86] suggest another type of bounding volume, which is a convex polyhedron consisting of the intersection of pairs of parallel planes. They call each of the plane pairs *slab*. Although these slabs allow a much tighter bounding volume, they have not been considered for *City Viewer*, as simple bounding boxes provide many advantages for implementation of both the ray tracer and the GPU renderer.

3.1.2 Bounding volume hierarchies

The bounding volumes explained before can be stored in nested hierarchies connected by nodes. Each node contains its bounding volume and several child nodes, except leaf nodes, which contain only their associated object. If a parent node is not affected by a ray intersection or is not inside the view frustum, the child nodes don't have to be considered. An example of 2D bounding volume hierarchies is given in figure 3.1.

Foley et al. [FvDFH96e] consider several methods for creating the hierarchies. While it seems intuitive to create the hierarchy manually during modeling, those hierarchies usually aim at controlling the object, instead of focusing on coherency. Goldsmith and Salmon [GS87] therefore developed a method to create the hierarchies automatically. This method already is quite similar to the *surface area heuristic* that will be introduced later.

Generally, bounding volume hierarchies are a bottom-up approach to object organiza-

[1] *City Viewer* uses bounding hyperrectangles, a generalization of bounding boxes or rectangles

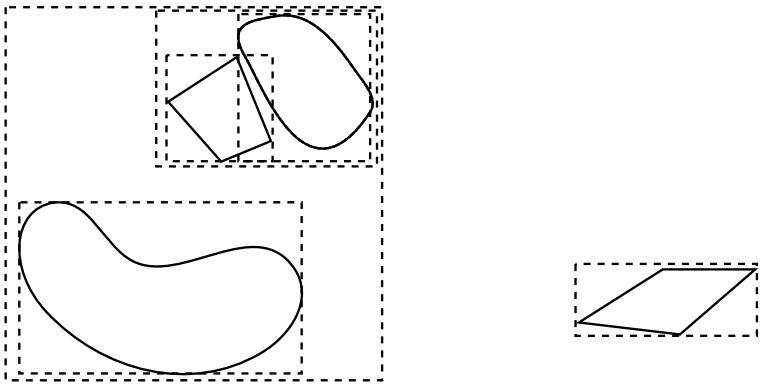


Figure 3.1: A simple 2D example of hierarchical bounding volumes. Children of bounding volumes may intersect.

tion, while the following techniques will use top-down logic. The bottom-up approach implies that child objects are guaranteed to be fully inside the parent bounding volume. However, objects may overlap a bounding volume without being a child of this volume.

3.1.3 Uniform spatial subdivision

This method divides the scene space into equally axis-aligned boxes. Each of this boxes contains a list of objects stored in it. In contrast to bounding volume hierarchies, an object stored in a box may not be fully inside this box. Subsequently, an object may be referenced by several boxes.

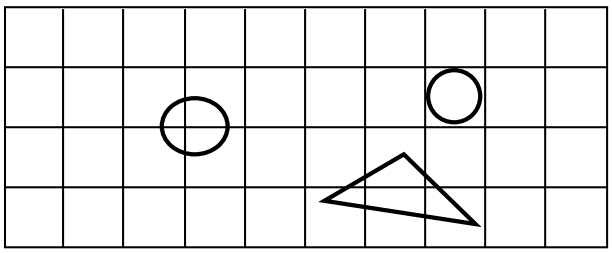


Figure 3.2: 2D uniform spatial subdivision

To find the nearest ray intersection, Foley et al. [FvDFH96e] suggest to use a modification of a line-drawing algorithm to find the partitions through which the ray passes. After testing all objects inside a box, traversal can be stopped if an intersection has been found[2].

[2]This is further complicated by the fact that an object may be part of several boxes. If the

3.1.4 Octree

Another option is to use an adaptive spatial subdivision method called octree. According to Foley et al. [FvDFH96d] octrees are based on the divide-and-conquer power of binary subdivision. They have been derived from quadtrees that are used for 2-dimensional data. An example of an quadtree is given in figure 3.3. An octree is a tree that either has no children or eight children. A parent node is subdivided in eight nodes of equal size[3] until a certain threshold is met, like a specific count of objects or polygons. Usually the node boxes are axis aligned.

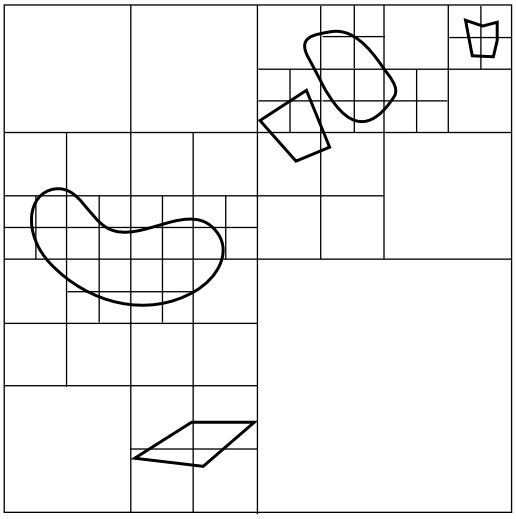


Figure 3.3: A quadtree subdividing 2D space

Octrees are used quite frequently for storing computer graphics data. They can be used well for frustum culling and ray tracing. It is possible to efficiently find a node's neighbors, which can be used for fast ray tracing.

3.1.5 Binary space-partitioning tree

Foley et al. [FvDFH96c] claim that a *binary space-partitioning (BSP)* tree fits very well to calculate visibility relationships of polygons. While the viewpoint can be chosen freely during runtime, the polygons themselves must be static. The latter is caused by the expensive preprocessing step necessary.

intersection is not inside the current box, traversal cannot be stopped, as the intersection may be far away.

[3]Variants of the octree allow different sizes for the nodes

A BSP tree is a binary tree that subdivides space by the use of hyperplanes. Each inner node contains a hyperplane that subdivides the space of the node into two half spaces. This is based on the work of Schumacker [SBGS69], who explains that a scene can be viewed as composed of clusters. In this case, the objects on one side of the plane form one cluster. This subdivision of space can be continued until either only one object is left or until a specific threshold is met.

In practice, the objects sorted into a BSP tree often are polygons. When building the tree, usually one of the polygons is selected as splitting plane. However, this introduces the problem of polygons intersecting this splitting plane, which therefore cannot be uniquely assigned a half space. This is usually solved by splitting the polygon into several polygons that do not intersect the splitting plane. Although this increases polygon count, it simplifies many of the algorithms that use the resulting BSP tree. To avoid massive polygon splitting, a polygon that causes the least amount of polygon splits should be chosen as splitting plane, while still providing a reasonably balanced tree. This is expensive to compute, but Foley et al. reason that a good approximation can be done by just testing a few triangles as split planes and choosing the best one.

The BSP tree can be used for a lot of algorithms, including back-to-front or front-to-back rendering[4], collision detection or ray tracing acceleration.

3.2 Kd-tree

A kd-tree[5] is a BSP tree with axis aligned splitting planes. This results in every node to be surrounded by a hyperrectangle. For 3d-trees the space occupied by the hyperrectangle (in this case a box) is also called *voxel*. Bentley [Ben75] first introduced this data structure. Due to the large application range, there are several variations of kd-trees. Some store one object defining the splitting plane at inner nodes[6], while others only store the splitting dimension and splitting value at each inner node (implying that leaves store all objects)[7]. When working with points as objects, the points usually are stored directly in inner nodes.

In computer graphics kd-trees usually store primitives like triangles (or objects consisting of primitives) instead of points. The primary computer graphics usage of kd-trees is space subdivision, as opposed to other usages like binary search presented in the original Bentley paper [Ben75]. Focus is put on finding (and culling) empty space in

[4]The back-to-front rendering can be used for a painter's algorithm without need for a z-buffer
[5]Short for k-dimensional tree
[6]Also called homogeneous kd-tree
[7]Known as non-homogeneous kd-tree or kd-trie

a scene, therefore reducing the amount of work necessary when tracing rays or rasterizing the scene. Culling empty space is important not only for a ray tracer[8], but also for doing efficient occlusion culling as presented later.

Using a kd-tree as data structure for *City Viewer* has been considered the best choice because of the following reasons:

Performance A hierarchical approach to space subdivision has significant performance advantages compared to uniform space subdivision

Flexibility As the name says, a kd-tree supports an arbitrary amount of dimensions without significant penalty for additional dimensions[9]. This is important due to *City Viewer* supporting 4-dimensional data for time dependency

Well-known Many ongoing papers and technologies use kd-trees[10]

Simplicity As the splitting planes are axis aligned, many algorithms can be vastly simplified, resulting in much better performance and less implementation issues

A disadvantage of using a kd- or BSP tree is the limitation to static, non-animated scenes. For *City Viewer*, this is not a problem, as city data usually does not include animations[11]. Additionally, there are approaches to circumvent this limitation:

- Real-time construction of kd-trees is a topic currently being researched [WH06, HMS06]

- Instead of using one large kd-tree for the whole scene, animated objects are associated with a local kd-tree. This local kd-tree can be transformed along with the enclosed object. If the object geometry itself changes, the local kd-tree can be discarded and rebuilt. Sometimes it may be beneficial to not transform the tree itself. For example, a ray entering a local kd-tree may be transformed instead to still enable certain optimizations specific to axis aligned split planes.

Handling dynamic scenes is intensively discussed by Wald[Wal04], but is out of scope for this thesis. However, *City Viewer*'s design allows incorporation of the techniques Wald suggests without much restructuring work.

This kd-tree implementation works on "objects" instead of handling primitives directly.

[8] An empty voxel is a cheap voxel to traverse
[9] This statement is targeted at the costs of the kd-tree itself, not the data contained
[10] This is also true for Octrees
[11] Shader effects that may be added later on like water waves usually can be handled like static geometry

This has the advantage that issues like splitting can be assigned to the objects. Also, objects can be of any geometry type, like points, lines, triangles or whole meshes[12]. Each object must support several operations like splitting or retrieving the amount of primitives that will result from a specific split. Also, each object must have an associated k-dimensional bounding hyperrectangle.

This section focuses on the construction of a kd-tree, as the actual usage of the tree is covered later on in the chapters 4 and 6.

3.2.1 Construction algorithm

The construction of the kd-tree is quite straightforward. As the whole scene is known in advance, the kd-tree is given an array of all objects included in the scene. Then a splitting dimension along with a splitting value is chosen based on all objects. The actual way of doing this depends on the splitting method the tree uses (splitting in the middle, splitting by median or using a *surface area heuristic*). After the splitting plane is known, two new arrays of objects are created, one for each side of the plane. All objects are then assigned to the appropriate side based on their bounding hyperrectangle. Objects intersecting the splitting plane are split along the plane[13].

This algorithm continues recursively until a termination criterion is met. Usual termination criteria are tree depth, object count or primitive count. Another termination criterion is provided by examining the *surface area heuristic* as shown below.

3.2.2 Common split methods

Basically, two decisions have to be made in each recursion step. First, the splitting dimension must be determined. Second, the split position along this dimension must be found.

Finding the splitting dimension can be done very naively by just basing it on the current node's depth, simply iterating through all dimensions equally. This may split along dimensions that actually don't have any extent anymore, therefore wasting a full traversal step. A better approach is to split along the dimension with largest extent. This produces more cube-like voxels according to Wald [Wal04], which is good for efficient ray tracing and occlusion queries. Many implementations of kd-trees that

[12] In practice, all objects currently used in *City Viewer* are triangle meshes
[13] Optionally, each child node may contain a reference to split objects

shall primarily be used for binary search use the dimension with highest variance as splitting dimension.

Choosing the actual splitting value for the given split dimension can be done several ways, too. A very simple method is to split at the median object regarding the split dimension. This results in quite balanced trees with about-equal depth of each leaf. Wald reasons that this is actually not optimal for a ray tracer[14]. A balanced tree is good for binary searches with each branch having the same access probability, resulting in about constant search duration. First, the probability of a ray hitting a branch depends heavily on the size of a voxel, which itself is dependent of the split location. But splitting at the median usually creates voxels of different size. Second, shooting a ray through the tree is comparable to a range search visiting several leaves. This range search includes frequent up- and down traversal of the tree. To optimize this range search, the tree should be constructed in a way that reduces the amount of traversals needed when tracing the ray. In other words, empty space voxels should be as close to the root as possible.

A better method to choose the splitting value for *City Viewer*'s purposes is to use the center of the current voxel. Together with choosing the split dimension based on extent this produces quite good results for usage in computer graphics. Some improvements can be made. For example, the split plane can be shifted towards the closest object's bounding box plane if it currently doesn't intersect any object.

3.2.3 Surface area heuristic

While the splitting techniques presented before do achieve acceptable results, they are by far not optimal. Wald [Wal04] suggests a heuristic method for predicting the cost of a split following ideas of Goldsmith and Salmon [GS87], MacDonald and Booth [MB89, MB90], and Subramanian [Sub90]. A cost prediction function estimates the cost of a split along a given split plane. The split is then done at the location with lowest cost. Possible advantages of choosing such a cost estimation function are shown in figure 3.4

A well researched cost estimation function is the *surface area heuristic (SAH)*, which has been presented by MacDonald and Booth [MB89, MB90]. This heuristic makes two assumptions. First, it assumes that rays are equally distributed in space, and second, it assumes that these rays cannot be blocked by scene objects. This allows to calculate the probability of a ray intersecting a voxel, given that it intersects the

[14]The same reasons can be applied to occlusion culling rendering

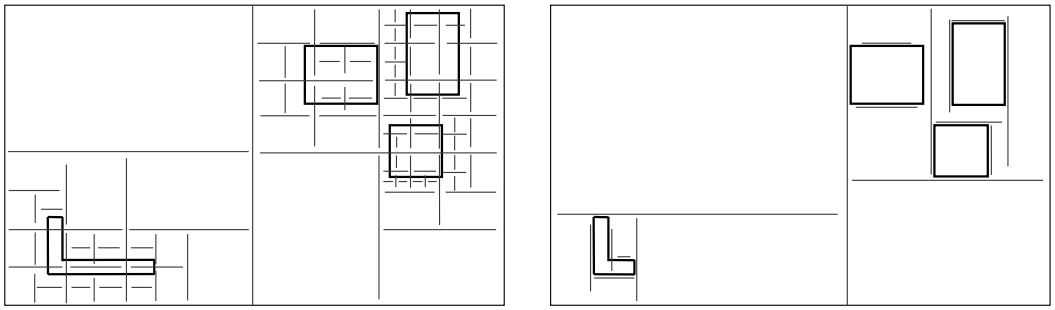


Figure 3.4: Left: an inefficient kd-tree built by splitting in the middle and alternating split dimension. Right: a typical SAH kd-tree which tries to cull empty space. The right kd-tree is much faster to traverse while approximating the geometry much better.

voxel's parent. The probabilities for the left and right voxels V_L and V_R to be hit (under the condition that the parent voxel V is hit) can then be calculated using the following formulae:

$$P(V_L|V) = \frac{SA(V_L)}{SA(V)} \tag{3.1}$$

$$P(V_R|V) = \frac{SA(V_R)}{SA(V)} \tag{3.2}$$

$SA(V)$ is a function that returns the surface area of the given voxel V. The calculation of the surface area is dependent on the dimension of the kd-tree. For 3-dimensional trees the surface area can be calculated as follows:

$$SA(V) = 2(V_{width}V_{depth} + V_{width}V_{height} + V_{depth}V_{height}) \tag{3.3}$$

Using these probabilities, the cost of a split can be calculated easily:

$$Cost_{split}(V_L, N_L, V_R, N_R) = C_{traversal} + C_{intersection}(P(V_L|V)N_L + P(V_R|V)N_R) \tag{3.4}$$

N_L and N_R specify the primitive count in the left and right subtree[15]. Two constants are introduced that allow tuning the heuristic to application specific needs. $C_{traversal}$ specifies the cost of a traversal, and therefore a fixed penalty for splitting a node. $C_{intersection}$ represents the cost of a ray triangle intersection. Both constants can be tuned to change shape and depth of an SAH kd-tree. The cost function can also be

[15]Optionally, N_L and N_R could be approximated by counting objects instead of primitives. This only works acceptably if the primitive count of objects doesn't differ greatly, which however happens frequently for deeper trees

used as termination criterion. Splitting is stopped once the cost of the cheapest split found is higher than not splitting the voxel at all. The cost of not splitting the voxel can simply be calculated as

$$Cost_{Unsplit}(V) = N_V C_{intersection}, \tag{3.5}$$

where N_V obviously is the primitive count of all objects. *City Viewer* uses the SAH termination criterion along with a maximum tree depth heuristic to control the tree shape.

Finding the best split plane

To find the best split plane for a given split dimension, all planes along this dimension that could possibly be the cheapest must be considered. The possible candidate positions are given by the vertex coordinates of all objects in the split dimension, as the cost function is continuous everywhere else [Hav00][16]. Once the candidate positions are known, the count of primitives resulting from each position is calculated. This is done by iterating over all objects and calculating the resulting primitive count if splitting at the given position. That way the cheapest split position can be found.

Instead of choosing a split dimension beforehand, this process is repeated for every dimension to find the cheapest split in all dimensions.

As this calculation is very expensive, the actual implementation applies several ideas to reduce runtime cost. The split candidates are sorted in the given dimension. This allows to keep a list of "right" and "left" objects[17]. Initially all objects are put into the right list. Iterating through the candidate list, the bounding hyperrectangles of the right objects are compared to the split positions. Objects that are fully left can immediately be put into the left list, and objects that are fully right or intersecting are kept in the right list. Intersecting objects are then asked to return the count of primitives left and right that will result if they are split at the current candidate. This algorithm saves a lot of calculations, as the left list does not need to be traversed at all, while the right list becomes smaller.

A very similar idea can be used to increase performance of primitive calculation of intersecting objects. Each object sorts its primitives based on the maximum coordinate

[16] This is only correct as this kd-tree implementation splits objects that intersect voxel boundaries, instead of keeping references to the unsplit objects in all intersected voxels. Not splitting the objects would require adding all intersections of primitive sides with voxel sides to the candidate list

[17] Actually no "left" list exists, as only the count of left primitives needs to be stored

in the given split dimension[18]. Instead of iterating through all primitives, a binary search can be done to find the first primitive that intersects the current split value. Additionally, the left boundary of the search range can be kept from the last request, as the split candidates are sorted.

3.2.4 Time

An important aspect when choosing the spatial subdivision scheme has been support for higher dimensions. The requirements for *City Viewer* include support for time dependent data. This is not to be confused with animation. Instead, each object may have a creation and termination date to support visualization of city development over centuries. This idea lends well to direct incorporation into the kd-tree, especially since cities usually grow at their boundaries. Therefore, *City Viewer* does not use a 3d-tree but a 4d-tree, although the geometry itself is still 3d[19]. The time dimension is called t. The term "surface area" is kept in this thesis, despite the voxel sides now being volumes.

As described above, the kd-tree only accesses the objects but doesn't directly know about primitives. This eases the introduction of time as fourth dimension. Basically, the kd-tree does not differ between dimensions[20]. All work regarding splitting or calculating hyperrectangles is delegated to the objects.

The formula for calculating the surface area for 4d voxels is different than the one for 3d voxels:

$$\begin{aligned} SA(V) = {} & 2V_{width}V_{height}V_{depth} + 2V_{width}V_{height}V_{duration} + \\ & 2V_{width}V_{depth}V_{duration} + 2V_{height}V_{depth}V_{duration} \end{aligned} \tag{3.6}$$

Remembering that the SAH is based on equally distributed rays, this formula doesn't seem to be optimal. When displaying a scene on the screen, only one point in time is displayed, no matter if using ray tracing or rasterization. Figure 3.5 visualizes the problem.

The formula is therefore modified to only include the surface area affected by rays

[18] It is sufficient to keep a sorted list of structures containing the min/max values of each primitive along the current dimension

[19] Real 4d geometry has been discussed shortly, but this introduces a bunch of memory and visualization problems if done really flexibly. However, *City Viewer*'s architecture can easily be extended to support key-frame animations which can be compared to restricted 4d geometry

[20] Except for calculating the surface area, as seen later

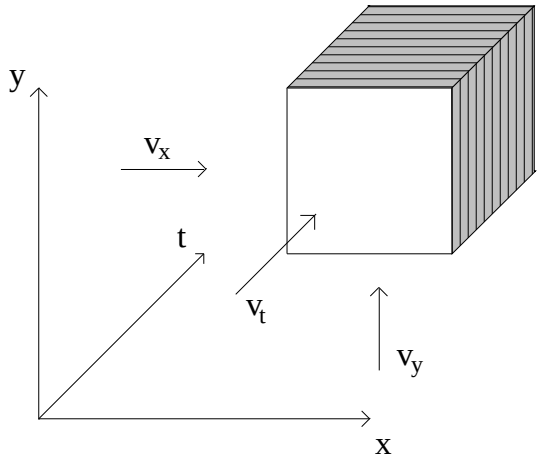


Figure 3.5: Visualization of a 4d-tree ignoring z dimension. Of all visible areas only the shaded areas may be used for SAH, as only ray directions v are valid that can be expressed by $v = (v_x, v_y, v_z, 0)^T$. This must be true because only one point in time is displayed on the screen at once.

with constant time coordinate:

$$SA(V) = 2V_{width}V_{height}V_{duration} + 2V_{width}V_{depth}V_{duration} + 2V_{height}V_{depth}V_{duration} \tag{3.7}$$

This excludes the two sides of the voxel hyperrectangle that would only be intersected by rays with time slope.

3.2.5 Splitting objects

As explained before, objects are responsible for splitting. If an object is split, it returns references to its "children", which are then used by the kd-tree[21]. The actual split algorithm depends on the split dimension.

Time splits are easy to implement, as no geometrical splits must be done. Two objects are returned with identical geometry. To save memory, the object geometry is not copied. Instead the child objects store a pointer to the original geometry object[22].

Geometrical splits[23] are harder to do. One approach would be to partition the primitives into left and right primitives and to copy intersecting primitives to both children.

[21] In practice this is slightly more complicated, as the geometry objects are owned by entities which reference to them. The geometry object therefore also notifies the owning entity of this change

[22] If a geometrical split follows, the geometry is copied from the original location, as both child objects may be subject to different geometrical splits

[23] Meaning splits at a x, y or z split value

The advantage of this method is that the actual primitive count may be kept low if the primitives are stored by reference. However, *City Viewer* does not store primitives by reference, as each geometry object is considered as independent. A global database of primitives would solve this problem, though. But this method has a serious disadvantage: it either means rendering primitives multiple times due to belonging to different objects, or using a flag for each primitive that specifies whether the primitive has already been rendered this frame. This is very hard to combine with other acceleration structures used by the renderer[24]. It is therefore improbable that the saved primitive draw calls outperform the second method of splitting objects, which has been implemented for *City Viewer*. This method splits primitives that intersect the split plane into several smaller primitives. As *City Viewer* just supports triangle primitives currently, only this split method is explained.

Splitting triangles along a plane

The algorithm employed to split triangles has been presented by Shirley et al [Shi05]. When splitting a triangle with vertices a, b, c along a plane it intersects, two vertices are on one side and one vertex is on the other side[25]. The algorithm first makes sure that vertex c is on one side of the plane and a and b on the other side. This is done by swapping the vertices[26]. That way, only two cases have to be considered: c being on the left or right side of the plane. Then the two intersection points A and B can be computed using linear algebra by plugging the parametric line equations of $\overline{ac}$ and $\overline{bc}$ into the plane equation. Finally, the resulting triangles $\triangle abA$, $\triangle bBA$ and $\triangle ABc$ can be added to the appropriate objects.

[24] Combination with the ray tracer is easier to do, as explained later. A solution fitting both needs is to change splitting methods once reaching the last rendering depth layer

[25] Exactly one vertex actually may lie directly on the plane. This vertex is considered to be on the side opposite to c (after swapping)

[26] Pairs of swaps need to be done to avoid normal flipping

Chapter 4

Rendering

This chapter explains several algorithms and technologies that are used by *City Viewer* to present the city scene on the screen. It assumes basic knowledge of computer graphics including the rendering pipeline, GPU programming and graphics APIs like OpenGL or DirectX. A very good API-independent reference for learning the basics of computer graphics is provided by the book "Computer Graphics: Principle and Practice" by Foley, van Dam, Feiner and Hughes [FvDFH96a].

After a short introduction to DirectX 10, *City Viewer*'s appearances model is presented. Then the way *City Viewer* stores geometry on the GPU is discussed, followed by the algorithms used to display the scene efficiently using a kd-tree.

4.1 DirectX 10 introduction

With the shipping of Microsoft Windows Vista, DirectX 10 [Mic08] became available to developers. While DirectInput and DirectSound have not received upgrades, Direct3D 10 has been introduced with significant changes. The most interesting change is the removal of the whole fixed function pipeline, implying movement of some functionality to the shader pipeline. Device capabilities bits have been removed to allow cleaner code development[1], which means support of most functionality of Direct3D 10 is guaranteed for every Direct3D 10 capable graphics card.

For performance reasons, Direct3D 10 does strict object validation when creating objects. Previous versions of Direct3D made many of these validation tests during the Draw calls. To obtain optimal performance, it's now even more important to reuse objects instead of recreating them frequently.

[1] Some tests may still be necessary, for example, hardware may not support all texture formats

4.1.1 Resources and views

In Direct3D 10, all resources are now derived from a generic buffer type. Instead of binding these resources directly as render/depth target or shader resource, views are bound. Direct3D 10 introduced the concept of views to allow access to resources from different stages of the pipeline. This includes the option to interpret data differently. This is very useful for creating geometry on the GPU: indices and vertices can be rendered directly to their appropriate buffers. In Direct3D 9 a vertex shader texture fetch using shader model 3 would be required, which usually is much slower. The view concept also allows to reinterpret data in another format, as long as the bit and component count per element is the same.

4.1.2 Effects and shader model 4

The effect model has received some changes in Direct3D 10. State values have been encapsulated in chunks to allow both easier distinction of states along with performance increase due to limited data transfers to the GPU. Usually the state blocks for each rendering pass are set in the effect file, but the application may also directly change state blocks.

All shaders used by Direct3D 10 must be specified in HLSL. The graphics programmer may no longer use assembler shaders, although the resulting assembler shader code can be debugged during runtime using tools like PIX.

The most interesting feature is shader model 4, which has been completely redesigned. A good introduction to this model is given by the DirectX SDK[Mic08]. Most limitations of earlier shader models have been removed. For example, shader model 4 allows unlimited constants and instructions (obviously capped by hardware limitations). This new shader model uses a common shader core which is accessible by all shader stages, along with additional unique functionality for each shader stage.

There are now three shader stages: the well known vertex and pixel shader stages have been extended by the new geometry shader stage. The (fully optional) geometry shader stage sits between the vertex and pixel shader stage. The input for the geometry shader is a whole primitive including all information available for a vertex, and if existing, even all information of adjacent vertices. The shader outputs a vertex stream[2] that is interpreted as primitive strip. A new strip can be started by calling the RestartStrip

[2]The type of stream must be specified for each shader. Valid streams are point, line and triangle streams

method of the stream. The count of vertices emitted that way can vary for each geometry shader call[3]. The output of the geometry shader may be passed to the pixel shader stage or to the stream output stage, which stores the generated primitives (expanded to primitive lists) in a vertex buffer.

4.2 Buffer collections

Each geometry object may have different information available for each vertex. Some objects only offer position data, while others also offer normals or texture coordinates. This information can either be stored in a single buffer or be distributed over several buffers. Also, *City Viewer* uses indexed rendering calls, which requires an index buffer.

To avoid code redundancy, *City Viewer* stores data in buffer collections. Each buffer collection holds one index buffer and one or several vertex buffers along with the necessary input layout[4]. By default all vertex data is stored in one vertex buffer. The associated buffers and input layout are bound using a single apply call, although flags are available to exclude buffers or the input layout.

The buffer collection is filled with vertices and indices in several calls. Once all vertices and indices have been added, the appropiate DirectX objects can be created. Generating the input layout requires a shader input signature for validation. This has been achieved by signature shaders, which contain no actual shading code.

4.3 Appearances model

City Viewer uses the concept of appearances to keep the main rendering code clean. These appearances should not be confused with *CityGML* appearances. Each geometry object is associated with an appearance that specifies how the object should be drawn. Each appearance is linked to a set of effect techniques for different purposes (for example rendering with and without using a shadow map). Abstracting the actual object rendering code with these appearances also allows for easy swapping of techniques. This can be done two ways:

- A single object may be linked to another appearance to change the style of

[3] A maximum number of vertices generated must be specified
[4] Direct3D 10 input layouts replace the vertex declarations used in earlier Direct3D versions

the object. For example, selecting an object may cause a special selection-appearance to be used[5]. This appearance can then apply special effects to highlight the object, like changing color or even shape of the object using other shaders.

- Instead an appearance instance (which is referenced by many objects) may be modified. For example, all objects drawn using a water appearance shall be made invisible or drawn without any transparency.

Each appearance instance may reference its own effect file. In practice, the effect file is specified for each class in *City Viewer*.

Not each draw call is done using an appearance. For example, rendering to a shadow map is specified in a default effect file, which is used for all object types.

4.4 Efficient rendering

In order to fully exploit the performance of current graphics hardware, certain requirements should be met:

- Geometry data should be uploaded to the graphics card using vertex and index buffers.

  Vertex sharing Vertex sharing between primitives should be employed to reduce vertex shader costs

  Coherency Coherency of indexed vertices is important, as the GPU processes the whole range from the first vertex being indexed to the last one for each draw call. A draw call indexing the first and last vertices in a vertex buffer of N vertices results in N vertices being transformed by the vertex shader, even if only a single triangle is rendered. A vertex sharing algorithm must consider this to avoid inefficient vertex sharing.

- Draw calls should be concatenated, if possible. Rendering N triangles at once is much faster than N times rendering one triangle.

- State changes should be kept to a minimum. Objects should be grouped by the state necessary for rendering them to avoid frequent state changes. This is

[5] *City Viewer* currently doesn't do this yet, but instead just changes the diffuse color of the object

especially true for changing shader, texture or input stream states.

These thoughts led to the introduction of draw batches, which will be covered below.

4.4.1 Draw batches

Each draw batch is composed of several geometry objects and rendered with a single draw call[6]. This requires that each draw batch has a single appearance, texture and material[7]. Also, all geometry objects inside a draw batch must have the same creation and destruction time. Additionally, each draw batch must be fully contained in a leaf kd-node to support occlusion culling as described later.

Generating draw batches

The most straightforward way is to create one draw batch for each geometry object after building the kd-tree. This automatically satisfies the constraint of a draw batch not overlapping kd-nodes, as the geometry objects have been split to fit kd-tree leaves. The draw batch is created based on material, texture and appearance information of the object. The vertices and indices of the object are added to the appropriate buffer collection[8], while storing the first index and index count in the draw batch.

4.4.2 Merging draw batches

After creating the draw batches, each one represents a single geometry object. Depending on the input data, this may be a single wall (or part of a wall because of kd-tree splitting). However, there may be several other wall objects in the same kd-tree node. Those may have the same appearance, texture and material along with the same creation and destruction time, which means their associated draw batches can be merged to a single draw batch that is responsible for rendering all of these geometries.

Quick merging can be done by sorting all batches of a kd-tree leaf by their respective properties like appearance and material. Then all batches that can be merged are

[6] An exception are selected objects. If a draw batch contains objects that are selected, each object is rendered on its own. The same applies to changing the appearance of single objects, which would either require recreation of the draw batch or single object rendering.

[7] Additional per-vertex data or texture atlases could be used to relax these constraints

[8] There are several buffer collections, as only some geometry objects have texture coordinates

next to each other. Each draw batch can be merged with all draw batches following that are compatible. Merging is simply done by considering two batches that are compatible. They differ only in the indices they are referencing[9]. As one draw call shall render both objects, the indices in the associated buffer collection have to be reordered such that both geometries' indices follow each other. While this is trivial, it has to be considered that the index list is global, meaning some draw batches now point to wrong indices. Therefore the start indices of all draw batches affected are fixed. As a linear search through all batches would be quite slow, a STL map mapping from start index to draw batch is used[10].

4.5 Rendering the scene graph

Once the draw batches have been created, rendering the kd-tree is straightforward. A simple method would be to traverse the tree front-to-back, resulting in many fragments of objects far away to be discarded by the depth test, which would otherwise be drawn (and overwritten afterward). Additionally, the traversal algorithm may make use of frustum culling as described by Clark [Cla76]. The basic idea is to make use of the hierarchical structure which is given by the kd-tree. As each child node is fully contained in its parent node, hierarchical culling can be performed. Traversal of a child node is neglected if its bounding hyperrectangle[11] is fully outside of the view frustum.

Frustum culling improves performance significantly if a large part of the scene is not in the current view frustum. However, very large parts of the scene can be inside the view frustum but still be invisible, because of other objects occluding them. The camera might be right before a wall separating the scene, but still the whole area behind the wall must be drawn. While many fragments will be discarded by the depth test, this is still a large waste of GPU processing time. The next section therefore introduces an algorithm that eliminates processing of the area behind the wall.

[9] They also differ in the geometry object list, but the list of the second batch is simply inserted at the end of the first batch

[10] Actually only fixing batches of the current node would be necessary because of the order the batches are created. But multithreading for creating the batches will break this, as well as some other algorithms for building the batches. The map ensures robustness in that case

[11] Clark suggests spheres as bounding volume

4.6 Occlusion culling

In order to detect such occlusion, current graphics hardware usually supports *occlusion queries*. Occlusion queries return the count of pixels that passed the depth test. After a query is started, primitives are drawn. The query is ended after all primitives of interest have been drawn. Then the query results are sent to the CPU, which can use this information in its decisions for further action. While the query itself is basically done for free[12], latency is introduced for transferring the result to the CPU.

As the bounding volume of an object by definition fully contains the object[13], it is possible to render the bounding volume using an occlusion query with disabled depth and color writes. Only if at least one pixel passes the depth test, the object included may be visible. This information is used for deciding whether to render the actual object. For high-polygon objects this can improve speed significantly.

This idea can also be applied to hierarchical structures like kd-trees. Using front-to-back rendering, a query is issued for every child node's bounding box[14]. If the node's bounding box is visible, the node is traversed, otherwise it is culled. This approach works quite well, as long as a large part of the scene is invisible.

As Bittner et al. [BWPP04] point out, waiting for the query result due to latency (along with the overhead of queries and box rendering) can actually decrease performance compared to frustum culling approaches, if a large part of the scene in the frustum is visible. They therefore suggest using a method called *Coherent Hierarchical Culling*, which is based on the following concepts:

Temporal coherency It is assumed that a node that has been visible in the last frame is still visible. For interior nodes this results in immediate traversal without issuing a query. For leafs, a query of the bounding box is started[15], while the geometry is rendered directly after the query without waiting for the query result

Interleaving Instead of waiting for the result of a query after issuing it, all queries are stored in a queue. Before traversing further nodes, the first query in the queue

[12] Not including the rendering cost, which usually is low, as queries are mainly used for low-poly objects like bounding volumes. Additionally, when rendering bounding volumes for queries no expensive shading and no writes to the depth and color buffer are used

[13] Modification by vertex shaders must be considered

[14] This section refers to boxes instead of hyperrectangles, as the graphics card does not know about time dimension

[15] A query on the geometry can be done, too. This however may mean that a leaf is classified invisible in frame N testing the geometry itself, while it is classified visible in frame $N+1$ due to testing the bounding box. Together with methods like focusing the shadow map on the visible scene, this can result in flickering shadows even while not moving the camera

is checked. If the results are ready and the node is visible, the corresponding node can be traversed or the associated geometry rendered

Each node contains information about the visibility status of the last frame it has been checked. If a node has not been checked in the last frame, it is considered invisible, otherwise a visibility flag determines visibility of the last frame. Storing visibility like that is efficient and easy to handle. Neither does the algorithm have to clear all node flags, nor does it have to keep a visible or invisible node list.

When examining a query result, visible nodes mark all parent nodes as visible, while invisible nodes don't. The first results in a "pull-down" of the queried kd-tree layer, while the second results in a "pull-up". This has been visualized in figure 4.1.

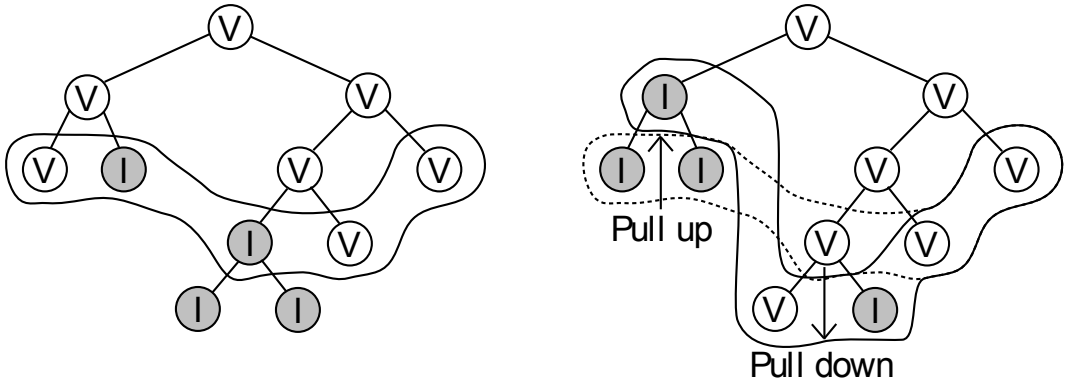


Figure 4.1: Occlusion queried layers of a kd-tree in frame $N-1$ (left) and N (right). Figure idea taken from [BWPP04]

In an optimal case, interleaving can fully hide the latency of queries according to Bittner et al. However, it may happen that no queries have finished, yet no nodes are anymore available for direct traversal. If that happens, the CPU needs to wait for the results of the first query in the queue, introducing a CPU stall (possibly followed by GPU starvation).

In their paper, Bittner et al. [BWPP04] suggest additional tweaks to improve performance:

Conservative visibility testing Instead of testing a node for visibility each frame, a visible node is assumed to stay visible for a certain time. This time can be a constant frame number or depend on camera movement or occlusion query success history

Approximation Nodes with only a few pixels visible can be assumed invisible. This has to be tuned carefully to avoid visible artifacts

Avoiding CPU stalls Instead of waiting for a query to return if no nodes remain to traverse, nodes with undecided visibility may be traversed without waiting for their query. The query is kept in the queue but the node is marked as traversed to avoid traversing a node twice

Implementation of this occlusion culling technique has vastly improved *City Viewer*'s performance when parts of the scene are invisible. On the other hand, performance didn't really suffer for the worst case of everything visible.

In the mean time, Mattausch, Bittner and Wimmer have presented a paper further improving this technique [MBW08]. Among other improvements, they focus on reducing state changes by batching previously visible and invisible nodes separately.

4.7 Compressing vertex buffers

As the memory needed for storing geometry data of a city scene can be quite large, compression should be used. With current graphics hardware, a vertex shader can be used for decompressing data of a vertex buffer that has been compressed on the CPU before uploading.

Vertex data like position, normals or texture coordinates is usually stored as 32-bit floating point data for each component. A position vector has four components, a normal vector has three components and a 2D texture coordinate vector has two components. For each vertex, this results in storing 16 bytes for position, 12 bytes for normal and 8 bytes for texture coordinates. Altogether each vertex uses 36 bytes uncompressed. After applying several simple and inexpensive compression methods, each vertex will only use 16 bytes without significant performance decrease or artifacts.

4.7.1 Position data

Position is usually stored in homogeneous coordinates, which consist of 4 components in 3-dimensional space. However, the position vector given by scenes is (almost) always homogenized, meaning the w component is equal to one. This effectively allows to simply store positions using 3 components. The vertex shader then simply initializes the w component to one.

While this does save 4 bytes per vertex without losing accuracy, a much better compression method can be used. Spatial coherency given by the kd-tree can be exploited

by storing relative coordinates instead of absolute coordinates. Each vertex can be expressed in relative coordinates with respect to the node containing the vertex. As the kd-tree used for the scene is quite deep and the volume contained quite small, it suffices to store each coordinate as unsigned 16 bit integer. This results in 8 bytes per vertex, as each element of a vertex buffer must be aligned to 4 bytes[16]. When rendering geometry, the vertex shader is given the current node extent, which allows it to decompress the vertex data easily.

4.7.2 Normal data

Normals can be stored in spherical coordinates instead of cartesian coordinates. This reduces each normal to two components. Additionally, using 16 bits for each component is sufficient. The current implementation therefore uses 16-bit floats for each component.

4.7.3 Texture coordinate data

The value range for storing texture coordinates is quite limited. While the maximum texture coordinate size on current hardware is 8192x8192, a 16-bit floating point variable[17] can store absolute integer values of up to 2048 without losing any accuracy. This suffices for most textures, whose size usually is below or equal to 2048x2048.

However, CityGML allows wrapped relative texture coordinates outside $[0; 1]$. Although used rarely, these may lose precision. Simply converting the coordinates to integer coordinates and using a modulo division by texture size on them can solve this. This may introduce artifacts for tiled textures that shall be contained several times in a polygon. In practice, these cases have not been encountered during development. The current implementation simply converts 32-bit floats to 16-bit floats without any additional actions. Optionally, 32-bit texture coordinates are used.

[16] To save memory, it has been tried to use two buffers for storing the position data to avoid alignment issues. The first buffer contained 2 bytes each for the x and y component, while the second stored 2 bytes for the z component. This resulted in vastly reduced performance, and has therefore been skipped

[17] 16-bit floats consist of 1 sign bit, 5 bytes exponent and 10 bytes mantissa

4.8 Transparency

City Viewer supports display of nonrefractive transparent materials. Rendering objects with transparent materials differs a great deal from opaque objects. Transparent objects are handled as explained by Foley et al.[FvDFH96b]. Nonrefractive transparency does not bend light rays when passing through a surface, which is of course not the case in reality, but is much easier to implement and usually provides good quality.

City Viewer handles transparent polygons by interpolating the shaded color values by the transmission coefficient $k_t \in [0;1]$ of the polygons. Considering two non-overlapping polygons with polygon 1 (partly) occluding polygon 2, the resulting color intensity can be calculated by

$$I_\lambda = (1 - k_{t1})\, I_{\lambda 1} + k_{t1} I_{\lambda 2}, \tag{4.1}$$

where k_{t1} represents the transparency of the polygon in front and $I_{\lambda 1}$ and $I_{\lambda 2}$ are the shaded illumination intensities of the front and back polygon. A transparency value of 1 means full transparency, while a transparency value of 0 represents a fully opaque object. In the implementation, interpolation is done by using alpha blending as supported by graphics cards.

This interpolation method obviously only works for polygons sorted back-to-front, as a different interpolation order results in completely different interpolated values. In a z-buffer-based system, polygons can be fed to the pipeline in any order. This means an opaque object behind an transparent object will not be drawn on the screen due to failing the depth test. Even putting all transparent polygons into a list and rendering them last with depth test disabled will not consider the order of the transparent polygons. *City Viewer* therefore sorts the list by depth before rendering the polygons. This produces the correct result, except that intersecting polygons will produce artifacts. These could be solved by sorting each pixel with a technique called *virtual pixel maps* as proposed by Mammen [Mam89].

Chapter 5

Shadows

Shadows are an essential part of high-quality computer generated images. Without shadows, scenes have an unrealistic look, as each natural scene has objects that cast shadows. There are several ways to add shadows to a rendered image, for example:

Ray tracing is a very high-quality method to render scenes. To calculate the shadow properties for each pixel, a ray is cast through the scene in the direction of every light source. If the ray intersects any object, no lighting from this light source is added to the pixel. To generate soft shadows, more expensive methods exist that cast many rays per pixel.

Shadow volumes are another way to generate precise sharp shadows. The general idea is to extrude the shadow caster geometry in light direction to infinity, which is called the shadow volume. This extruded geometry is then rendered into the stencil buffer of the frame buffer, with front facing polygons increasing the stencil count and back facing polygons decreasing the stencil count. If the stencil count for a pixel is non-zero, the pixel is in shadow, otherwise it is not. Issues arise when the camera position is inside the shadow volume, which requires more complex tests. The easiest method suited to solve these, *depth fail*, is patented by *Creative Labs* [BS02].

Shadow maps are probably the most common way nowadays to generate real-time shadows for directional and spot lights. The scene, as seen by the light, is rendered to a texture (shadow map), storing the depth values. Afterward, the scene is rendered again from camera view and using shadow map information.

Our application has several requirements regarding shadows:

- Support of large scenes

- Support of high-polygon scenes
- High-quality shadows
- Real-time calculation

Sadly, none of above approaches meet each requirement.

Ray tracing, even if only used for calculating shadows, does not support high frame rates on standard consumer hardware, although newer research proved at least low frame rate interactive rendering using *SSE* and multi-core processors [Wal04].

On the other hand, shadow volumes work nicely for many shadow applications, however they are not really suited for high-polygon scenes. The main problem using them is the high amount of fill rate required to render the extruded polygons, along with patent issues.

Shadow maps seem to be the most appropriate way to implement shadows in our case. However, they fail at meeting the quality requirement. Shadows generated by shadow maps often suffer from aliasing artifacts. Although using higher resolution shadow maps helps to increase quality, the resolution required to appropriately represent the shadows of a whole city is not affordable. There are several works researching methods to increase shadow map quality using more than one shadow map or using perspective matrices that increase detail of the shadow map closer to the camera. None of the methods really achieved the quality intended for *City Viewer*, but shadow maps are the best choice for the application and therefore the default technique for displaying shadows.

5.1 Shadow maps

The idea behind shadow maps is quite simple. Basically, the scene is rendered as it is seen by the light. Instead of color information only depth information is stored in a texture. This texture is called *shadow map* (visualized in figure 5.1). This depth information can then be used in the eye render pass as follows: For each rendered pixel, the z coordinate of the pixel's world position transformed by the light view matrix is compared to the depth value stored in the shadow map. If the depth value of the shadow map is less, the pixel is in shadow, otherwise it is not. As this approach is image based, it suffers from aliasing artifacts, especially in large scenes.

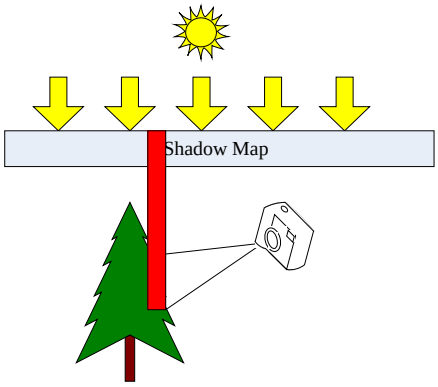


Figure 5.1: A common shadow map setting. One texel of the shadow map is marked. Figure idea taken from [Sch05]

5.1.1 Self-Shadowing

A simple implementation of shadow mapping suffers from strong artifacts due to self-shadowing of primitives, as seen in figure 5.2. This is one of the main problems when using shadow maps. Self-shadowing is immediately visible to the user, and therefore should definitely be avoided. An easy approach to reducing these artifacts is to add a depth bias to the shadow map. This is visualized in figure 5.5. This bias must not be too large, otherwise shadows will be missing when the shadow caster is close to the shadow receiver.

Another popular way to avoid self-shadowing is to ignore front facing polygons when rendering the shadow map, however this introduces artifacts in several cases like figure 5.3. *City Viewer* therefore does not use this method but a bias.

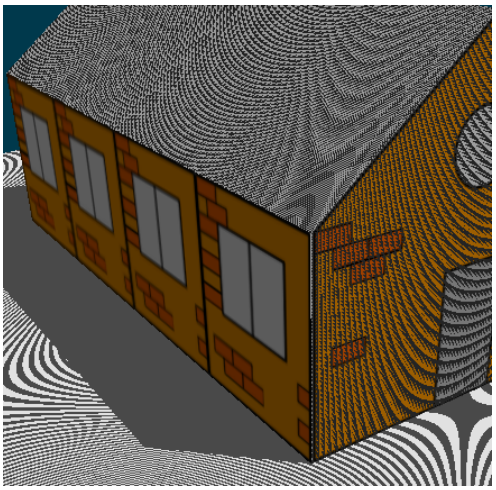


Figure 5.2: Shadow map self-shadowing artifacts

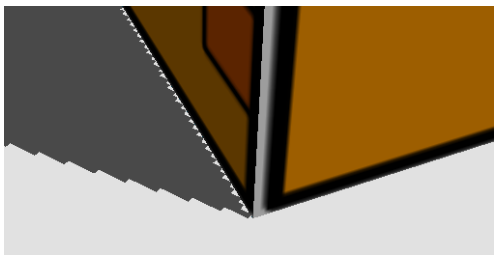


Figure 5.3: Shadow map including backfaces only

Setting the bias

Trivially, a constant bias could be added to the depth value before storing it in the depth buffer of the shadow map. However, this makes it difficult to balance the bias, as primitives with a high depth slope compared to the light direction need a much larger bias than those that are orthogonal to the light direction vector. This can be seen in figure 5.4.

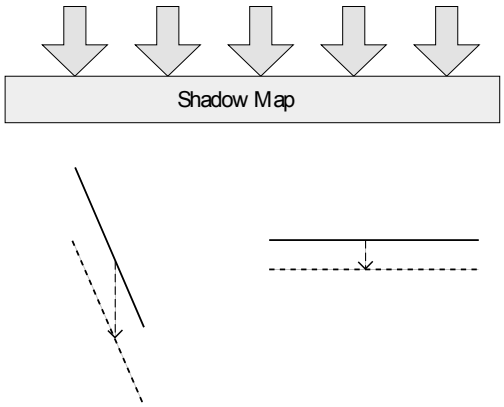


Figure 5.4: Two lines with different depth slopes compared to the light direction. The left line needs a much larger depth bias than the right line, as its depth slope is much larger

Usually 3D graphics APIs (in this case Direct3D 10) therefore offer a way to specify several constants which specify the total depth bias added to a primitive:

DepthBias A constant value that is added to the calculated depth value

SlopeScaledDepthBias This constant is multiplied with the polygon's maximum depth slope after transformation and added to the depth bias

DepthClamp Maximum depth bias that may be added to the depth value

The total bias is then calculated by

$$Bias = \min\left(DepthBias + SlopeScaledDepthBias \cdot MaxDepthSlope, DepthClamp\right) \tag{5.1}$$

This equation only applies to positive depth clamp values. For negative clamp values

(which are not useful for shadow mapping, though), the following equation applies:

$$Bias = \max\left(DepthBias + SlopeScaledDepthBias \cdot MaxDepthSlope, DepthClamp\right) \tag{5.2}$$

The bias calculated this way is applied by Direct3D 10 to the vertices after clipping. However, the depth value calculated using the bias cannot be read by the pixel shader. If the pixel shader needs to output the modified depth value to the render target instead of the depth buffer, this algorithm has to be implemented manually.

5.1.2 Aliasing

Another problem when using shadow maps is projection and perspective aliasing.

Projection aliasing happens when the camera view angle differs from the shadow map view angle, as it is often the case when walking through a scene with sunshine, which can be seen in figure 5.1. The different angle accounts for a large difference in the amount of texels associated with a specific scene region. Figure 5.1 shows that only a few texels of the shadow map are associated with the visible part of the tree, while a large part of the screen is occluded by the tree.

This is worsened by perspective aliasing, which is created by the perspective view used for displaying the scene to the user. Objects that are closer to the camera are larger on the screen than objects far away. However, the shadow map usually does not account for this, which results in a low resolution and therefore undersampling of shadows close to the camera, while shadows in the distance are often oversampled.

Perspective aliasing "scales" the error introduced by projection aliasing. Figure 5.3 shows these artifacts as "jagged boundaries".

5.2 Variance shadow maps

5.2.1 Idea

Variance shadow maps try to reduce the artifacts introduced by shadow map aliasing. The idea is to interpolate the shadow boundaries as can be seen in figure 5.6. However, you cannot directly interpolate between texels of a standard shadow map, as the

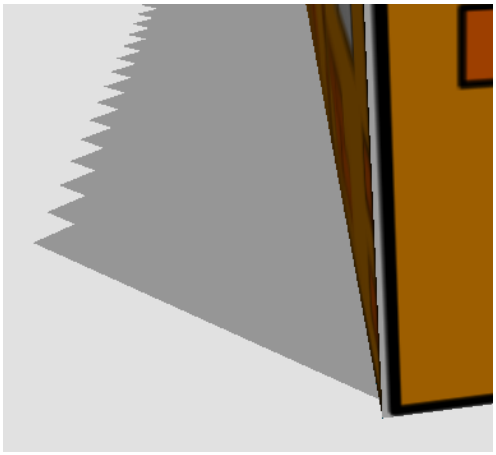


Figure 5.5: Low resolution standard shadow map

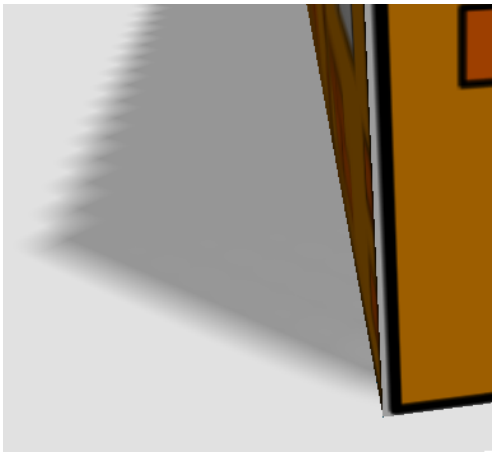


Figure 5.6: Low resolution variance shadow map

shadow map stores depth values. Linear interpolation would be incorrect for boundary edges of objects, as can be seen in figure 5.7. However, Donnelly and Lauritzen

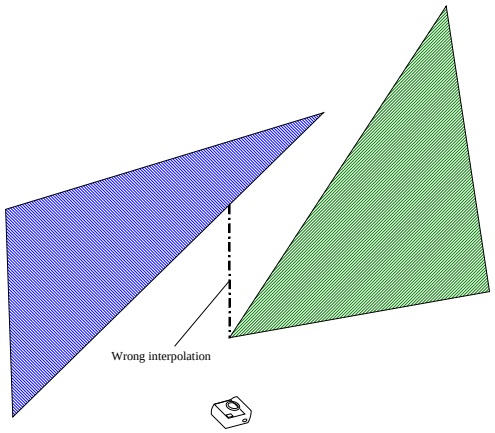


Figure 5.7: Incorrect depth interpolation

introduced the variance shadow map algorithm [DL06]. *Variance shadow maps* don't store depth values, but the mean and squared mean of a depth distribution. Using these two values the variance over a specific filter region can be computed. The variance can then be used to compute an upper bound of the fraction of occlusion of a fragment. As variance shadow maps store moments of depth distributions, this allows use of standard filtering techniques available on current hardware, like mip mapping or anisotropic filtering. Variance shadow maps don't make any assumption about shadow map projection, and are therefore suited to be combined with perspective shadow maps described below.

5.2.2 Implementation

The implementation of variance shadow maps is comparable to standard shadow maps. Instead of storing depth values in a depth texture containing one component per fragment, the depth and squared depth are stored in a two component texture. This texture can then be subject to additional filters, like Gauss filtering or mip mapping.

However, depth biasing can't be applied like explained in section 5.1.1, as the shadow map is stored inside a color buffer instead of a depth buffer. The bias calculated by Direct3D is not readable in the pixel shader, and therefore can't be applied to the output shadow map. It only affects the depth values stored in the depth buffer. This has been solved by just using a constant bias stored as shader constant. Of course, slope scaled biasing could be added manually, too.

When rendering the scene on the screen, the moments M_1 and M_2 from the filtered texture must be retrieved and used to calculate the mean μ and the variance σ^2:

$$M_1 = E(x) \tag{5.3}$$
$$M_2 = E(x^2) \tag{5.4}$$
$$\mu = E(x) = M_1 \tag{5.5}$$
$$\sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2 \tag{5.6}$$

Donnelly and Lauritzen show that by using Chebychev's inequality (one-tailed version), an approximation of the fraction of pixels over a filter region that will fail a depth comparison with fixed depth t can be calculated. For $t > \mu$

$$P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{5.7}$$

$p_{max}(t)$ can then be used as approximation of the amount of occlusion for the current pixel.

5.2.3 Light bleeding

Due to the way variance shadow maps interpolate, it can happen that areas that actually are in shadow are incorrectly classified as lighted. This can be seen in figure 5.8. These artifacts happen when a scene has high depth complexity. The high depth complexity leads to a sharp change in variance.

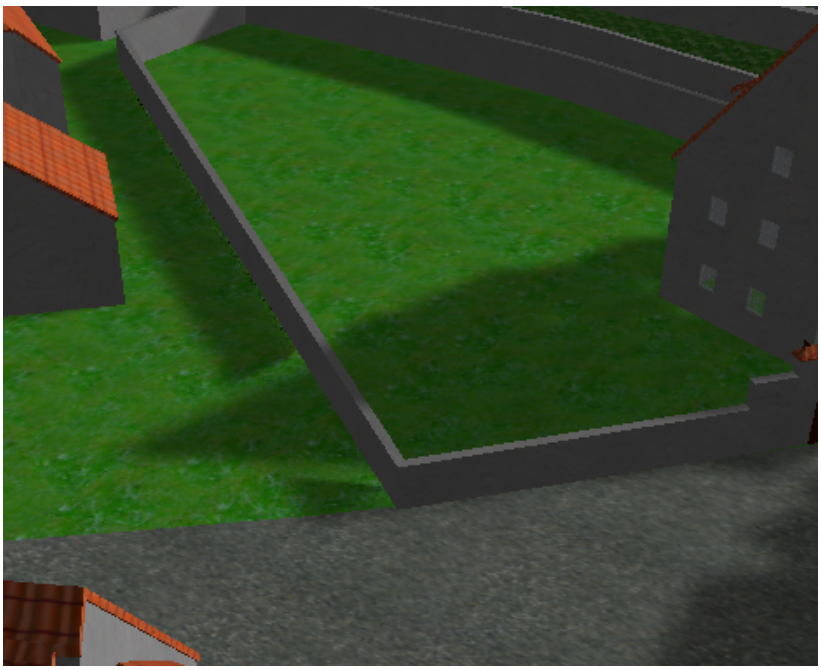


Figure 5.8: Variance shadow maps suffer from light bleeding, as seen behind the wall

5.3 Image based Gauss smoothing

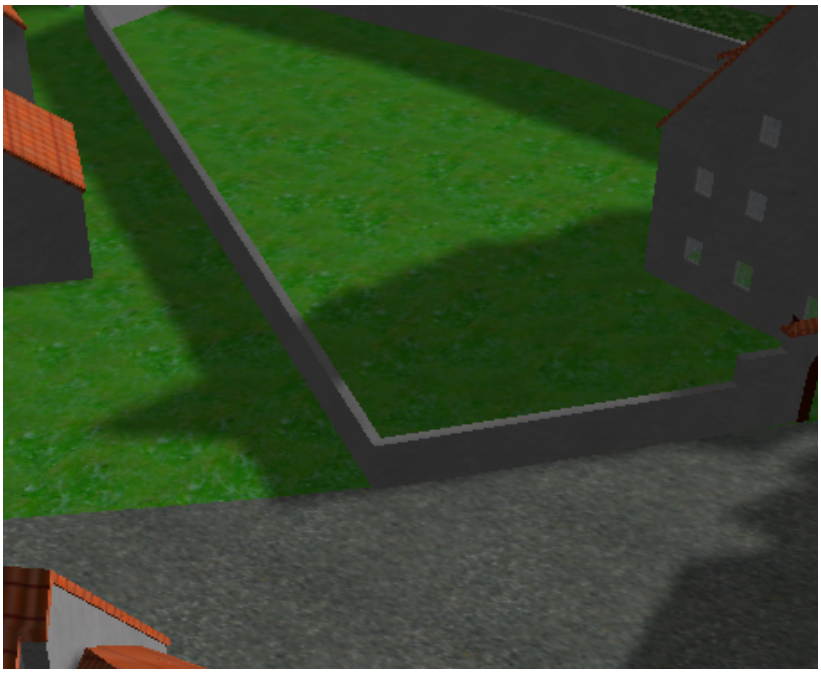


Figure 5.9: Image based Gauss filtering. No light bleeding artifacts as in figure 5.8

Although *Variance Shadow Maps* allow to interpolate between shadow map texels, the resulting light bleeding artifacts have been conceived prohibitive for *City Viewer*. As an alternative, image based smoothing may be used. The advantage of an image based approach is that it can be applied to other shadow techniques, too. It can be

applied to shadows generated by shadow volumes or ray casting just as it can for shadows generated by shadow maps. Therefore a well designed image based smoother can easily be reused for other shadow visualization techniques.

Of course, it must be made sure that the shadows are only smoothed where they should be. This means that errors like seen in figure 5.10 must be avoided using scene information. Also smoothing must be based on distance to camera, otherwise shadows will appear too fuzzy.

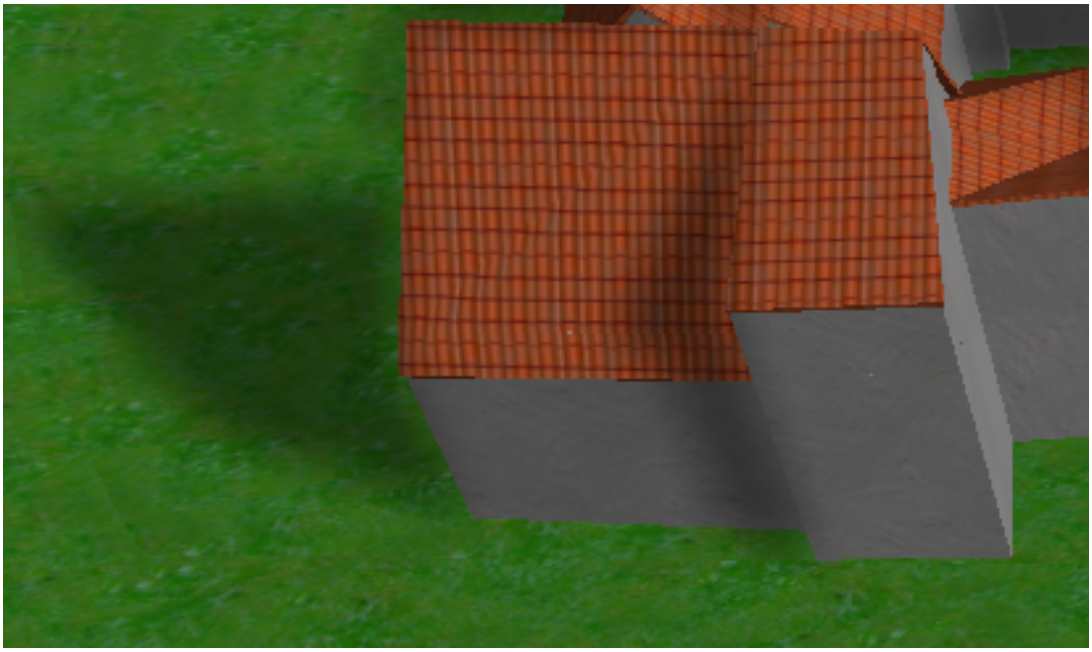


Figure 5.10: Shadow incorrectly smoothed along building edges using naive implementation

The general algorithm idea is as follows:

- Render shadow map as usual
- Apply shadow map to scene and render the applied shadow map to texture as seen by the camera
- Apply Gauss filter to texture
- Render scene as seen by the camera, and use the smoothed texture as (image based) shadow lookup

5.3.1 Gauss image filtering

Generally, Gauss filters in image processing are used for smoothing images. While large structures remain, small structures are removed. This basically is what is needed for removing shadow map aliasing artifacts. A two dimensional gauss filter can be created based on the following formula:

$$h(x,y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5.8}$$

σ specifies the standard deviation of a Gaussian distribution. This parameter controls smoothing strength. In practice, a matrix with size $N \times N$ is used for specifying the Gauss filter kernel. x and y then specify the distance from the center of the kernel in each dimension, and the result $h(x,y)$ is the weight of the color at pixel (X/Y)[1] to the new pixel color.

To improve the speed of the smoothing operation for larger kernel sizes, the Gauss filter can be separated into a $N \times 1$ and a $1 \times N$ filter. The whole smoothing operation is then done by convoluting the two 1D Gauss filters.

5.3.2 Using scene information

The actual filter being used must use scene information to avoid smoothing areas that are not connected to each other. For example, in figure 5.10 the grass behind the wall is partly being lighted, as the Gauss filter smoothed the lighted area of the wall left to the grass area.

While the shadow image is being smoothed, the scene already has been rendered for the initial fill of the shadow image. Therefore access to the depth buffer is possible. The Gauss filter can use the depth information to reproduce the world position of each fragment it is smoothing. That way, when iterating through the kernel matrix, elements whose world position is too far away from the central element world position can be ignored. This threshold value can either be a constant or subject to modification by the distance to the camera. The further away from the camera, the lower the threshold should be, as when the camera is right in front of an object, smoothing should only occur between several centimeters in world coordinates, while further away the threshold may easily be a meter.

The other issue that has to be taken care of, is choosing σ, which sets the actual smooth

[1] relative to the current pixel being modified

strength. Shadows that are far away may not be smoothed as strong as shadows close to the camera.

σ and the maximum distance threshold may not be confused. While the first chooses the smoothing strength of the Gauss filter in image space, the second considers world space information. Both have to be tuned independently to achieve a good looking result.

5.3.3 Conclusion

Smoothing in image space can produce very realistic soft shadows, given that the shadow map aliasing is not too strong. However, if the camera is close to an aliased shadow map edge, the filter size necessary for smoothing may be prohibitive for good performance. Variance shadow maps adapt better to this, though light bleeding may actually create worse artifacts than aliasing.

In conjunction with ray tracing shadows, Gauss smoothing in image space really shines. While the soft shadows are fake, high-quality soft shadows can be achieved in real-time. By tuning the smoothing constants the softness of the shadow edges can easily be modified, thus allowing to fake area lights of different size.

5.4 Perspective shadow mapping

There are several approaches to decrease perspective aliasing which involve adding a perspective transformation to shadow maps to increase shadow map resolution close to the viewer. This section will introduce some of those methods. *City Viewer* itself uses either standard uniform shadow maps or *Extended Perspective Shadow Maps* (XPSM).

5.4.1 Perspective shadow maps

As described in figure 5.1.2, perspective aliasing is a major problem when using shadow maps. Objects that are close to the user are projected to a large part of the screen, while objects far away will be quite small. However, uniform shadow maps do not take care of this.

One approach to decrease perspective aliasing is to increase shadow map detail close to the camera, while decreasing shadow map detail for objects far away. Stamminger and Drettakis therefore tried to reduce perspective aliasing using a method called *Perspective Shadow Maps (PSM)* [SD02]. Their approach generates the shadow map in post-perspective space of the camera. In this space objects close to the camera are larger than objects far away, which is the desired effect.

An illustration of this approach can be seen in figure 5.11.

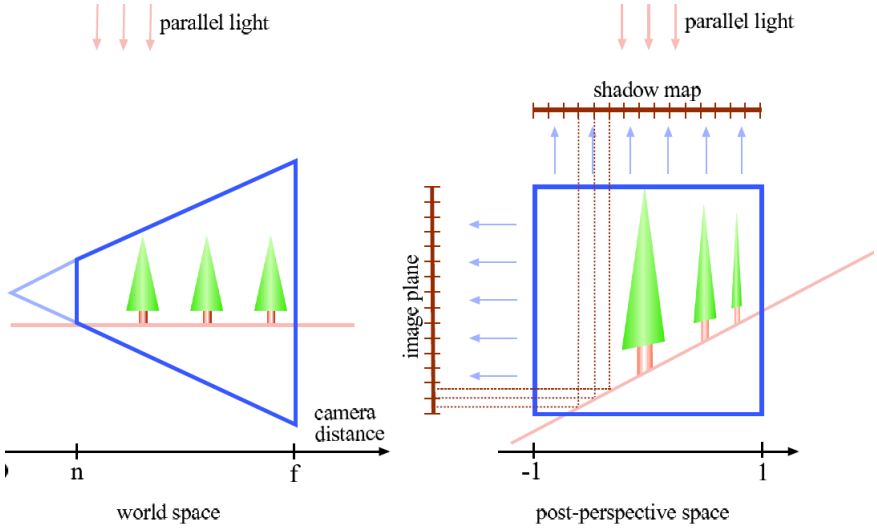


Figure 5.11: Directional light source applied to a scene (left). The shadow map is generated in post-perspective space (right). Figure taken from [SD02]

The general algorithm therefore is to transform the scene (including light source) to the post-perspective space of the camera. Then, the scene is rendered from the (transformed) light source to the unit cube. As the shadow map is generated after perspective projection of the scene, in most cases perspective aliasing is reduced by a large amount. Best results are achieved with the light direction being orthogonal to the view direction.

However, generating the shadow map in post-perspective space does have some issues that need to be handled. On the one hand, directional lights can turn to point lights in post-perspective space and vice versa. For example, a directional light from behind the camera will turn to a "inverted" point light, which needs special treatment by reversing the shadow map depth test. On the other hand, it must be guaranteed that all objects that may cast shadows to the visible scene are included when generating the shadow map. This causes problems when these objects are behind the camera, as those are projected beyond the infinity plane. Stamminger and Drettakis propose fixing that issue by shifting back the camera until all caster objects are in front. This shiftback only applies while creating the shadow map. However, this again increases

the remaining aliasing artifacts, especially for scenes that already tend to converge to uniform shadow maps.

Because of those implementation issues *PSMs* have not been included in *City Viewer*. Still, the idea of *PSMs* spawned several other methods to generate shadow maps, which will be described below.

5.4.2 Light space perspective shadow maps

Wimmer, Scherzer and Purgathofer introduced a new technique called *light space perspective shadow maps* [WSP04] to fix the issues caused by perspective shadow maps. Most of the problems, like light types changing and inclusion of shadow casters behind the camera, have their seeds in the camera perspective projection before creating the shadow map. This results in complicated implementations. However, they argue that the perspective projection that warps the shadow map does not have to be tied to the view frustum. They made two observations:

- Any arbitrary projection transformation can be used to warp the shadow map
- As the idea is to change the shadow map pixel distribution, the warp only has to affect the shadow map plane and not the perpendicular axis

This transformation, in contrast to PSMs, does not change type and direction of light sources. Additionally, the problems involving shadow casters behind the camera are fixed, as no singularities affect the shadow map generation. However, the method also doesn't take care of projection aliasing.

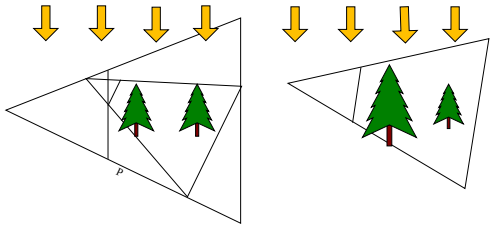


Figure 5.12: On the left: frustum showing perspective transform P and view frustum, with light direction being parallel to the near and far plane of P. Right: after applying P. Figure idea taken from [WSP04]

Algorithm idea

1. Focus shadow map on convex body B that includes the view frustum and all objects casting visible shadows

2. Enclose B with perspective frustum P as can be seen in figure 5.12, using a view vector that is parallel to the shadow map plane

3. Choose free parameter n of P, which is the distance of the projection reference point p to the near plane of P. This parameter controls the warping strength.

4. Use P for generating and reading the shadow map.

5.4.3 Extended perspective shadow maps

Vladislav Gusev proposes a two step method [Gus07] for calculating the shadow map matrix. The first step is to find the optimal warping effect followed by finding an affine transformation that doesn't affect this optimal warping and transforms the warped space to device normalized coordinates.

Gusev points out that you can apply any arbitrary transformation to light space[2], as long as all points along a light ray are projected to the same point (X'', Y'')[3] of the shadow map. The transformation applied can be described as follows:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ W' \end{pmatrix} = \begin{pmatrix} X \\ Y \\ t \\ 1 \end{pmatrix} \cdot \begin{pmatrix} Ax & Ay & Az & Aw \\ Bx & By & Bz & Bw \\ Cx & Cy & Cz & Cw \\ Dx & Dy & Dz & 1 \end{pmatrix} \tag{5.9}$$

$$\begin{pmatrix} X'' \\ Y'' \end{pmatrix} = \frac{\vec{A}xy \cdot x + \vec{B}xy \cdot Y + \vec{C}xy \cdot t + \vec{D}xy}{Aw \cdot X + Bw \cdot Y + Cw \cdot t + 1} \tag{5.10}$$

As X'' and Y'' must be independent of t in order to satisfy the requirement of all light ray points being projected to the same point on the shadow map, two constraints can

[2]Light space is defined by Gusev as light direction parallel to the z axis and the viewer origin translated to (0, 0, 0)
[3]cartesian coordinates

be introduced and used to build the needed arbitrary transformation formula:

$$\vec{C}xy = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{5.11}$$

$$Cw = 0 \tag{5.12}$$

$$\begin{pmatrix} X'' \\ Y'' \end{pmatrix} = (\vec{A}xy \cdot X + \vec{B}xy \cdot Y + \vec{D}xy) \cdot \frac{1}{Aw \cdot X + Bw \cdot Y + 1} \tag{5.13}$$

Therefore the projection vector is $(Aw, Bw, 0, 1)^T$.

To find the direction of the projection vector, simply the direction of the camera view (in light view space) must be projected onto the xy plane. Finding the length of the projection vector (which defines the warping strength) isn't that straightforward, however. Objects behind the camera that cast shadows onto the currently visible scene must be considered, as singularity issues may arise when projecting them. Further details can be read in Gusev's XPSM paper [Gus07], which describes the whole algorithm in detail.

In practice, XPSM's proved quite useful and robust while tuning the shadow algorithms of *City Viewer*. Still, projection aliasing problems were not addressed.

5.5 Cascaded Shadow Maps

While PSMs, LSPSMs and XPSMs try to reduce perspective aliasing by applying a projection transformation to the shadow map, *Cascaded Shadow Maps (CSM)* use a different approach. Perspective aliasing is countered by creating several shadow maps, each one being responsible for a different part of the scene.

One idea of using multiple shadow maps would be to statically divide the scene into several parts, and rendering each of those parts to a shadow map. However, this static approach is quite limited. If the camera was centered in one of those parts, a large part of the shadow map would be wasted. Also, some of the shadow maps would be completely unused if their scene parts were fully behind the camera.

Cascaded Shadow Maps therefore dynamically select the parts of the scene each shadow map represents. Each shadow map shall represent a part of the view frustum. The parts are selected by different near and far clip planes of the frustum, therefore having one shadow map for near scene objects and one for far away objects. Additionally,

more shadow maps can be used for objects in between.

Basically, *Cascaded Shadow Maps* have the same goal as PSM methods: increasing shadow map resolution close to the viewer while decreasing it far away. However, they do this in a discrete way, contrary to PSMs.

Selecting the near and far plane for each frustum should not be hard coded. Instead, it should be tried to provide the same aliasing error on the whole screen. That way, visible changes in the shadow quality can be avoided. Dimitrov [Dim07] shows that for a large number N of splits, the split planes should be located at:

$$z_i = n \left(\frac{f}{n} \right)^{\frac{i}{N}} \tag{5.14}$$

Typically N is small, though. This exposes the user to the split planes. To avoid this, Dimitrov [Dim07] adds a linear term:

$$z_i = \lambda n \left(\frac{f}{n} \right)^{\frac{i}{N}} + (1 - \lambda) \left(n + \left(\frac{i}{N} \right) (f - n) \right) \tag{5.15}$$

As the shadow map frusta are now known, each shadow map must focus on its frustum. For each frustum the corner points are calculated and transformed to the orthogonal light view. Then the bounding box of the transformed corner points is calculated. Using this bounding box an offset and scale matrix can be calculated to focus the shadow map on the frustum.

Afterward N shadow maps are rendered. Usually, the render target is a texture array. That way, the pixel shader can directly sample the correct texture. To compute the index of the texture, the z-far distances of each frustum are uploaded as shader constants. The pixel shader then simply compares the current z value to these constants to find the appropriate shadow map texture index.

Otherwise, everything works as usual. The technique can easily be combined with other technologies like *Variance Shadow Maps* or *Extended Perspective shadow Maps*. Combination with perspective shadow map techniques does not produce a vast improvement, though, as both improve the same aliasing problems.

5.6 Refining shadow maps using raytracing

Using shadow maps, shadows often have poor quality, as seen in figure 5.5. Therefore, a hybrid approach has been added to *City Viewer*. The idea is to apply the shadow map to the scene as usual. Afterward, the poor quality shadow edges are refined using raytracing. That way it is only necessary to cast rays for parts of the screen, which is much faster than raytracing the whole screen.

This is the general algorithm idea:

1. Calculate shadow map (as seen by light)
2. Apply shadow map to scene and render scene to texture (as seen by eye).
    a) Apply shadow map to scene as seen by the eye
    b) Use special shader that writes primitive ID and flag specifying if pixel needs to be refined to texture
3. Download texture to CPU
4. For each pixel which needs to be refined:
    a) Cast ray in light direction and write result to texture
5. Upload resulting texture to GPU
6. Render scene as seen by the eye using original shaders and looking up shadow information in the refined texture

The primitive ID is used to find the world position of the pixel (using ray-primitive intersection). This could also be done by exploiting depth buffer information, however this introduces further accuracy problems. This approach was therefore skipped, although it was slightly faster. Another benefit of storing the primitive ID is the option to add other features like reflections using the ray tracer, as the ID allows us to access surface information instead of only position information. Also, the ID approach allows to use transparent objects, as the ray tracer can access material information. The general idea of using an item buffer to find the closest object at a given pixel has been described by Weghorst, Hooper and Greenberg [WHG84].

Basically, the first ray generation is casted by the GPU using rasterization methods, along with some second generation rays for parts of the shadows, while all other

ray generations are handled by the CPU. This hybrid approach allows for quite fast implementations of high-quality shadows for large-scale and high-polygon scenes.

In *City Viewer*'s implementation, the refined texture has 4 components with 8 bits each. The RGB components hold optional color data that will simply be added to the GPU calculated color data, while the Alpha component holds shadow information with 0 representing no shadow and 255 representing full shadow. This approach only supports one light source which is considered appropriate for *City Viewer*'s uses. Support for several lights or additional information can easily be added by using several textures or increasing bit count.

5.6.1 Finding shadow edges

In order to refine the (projected) shadow map, the edges of the shadows projected into eye view have to be found. This has been achieved using variance shadow maps described before. Whenever the variance calculated is not zero, the texel is marked as to be refined.

Of course, this approach has some limitations. It is still based on the quality of the shadow map. Therefore, if the shadow map completely misses a hole in a shadow casting object, this hole will still be missing after refinement. Also, several different artifacts can occur under extreme circumstances.

To avoid these artifacts, one can filter the variance shadow map before applying it, for example using a Gauss filter. If all artifacts need to be avoided, this would result in shadow rays being cast for the whole screen. Although this can still be done interactively, doing this does affect frame rate quite a bit. Finding that trade-off is application specific or can be left to the user.

Another way to decrease raytracing workload is to avoid refining areas that are far away from the eye. If this threshold is chosen wisely, introduced artifacts are almost invisible to the human eye, but the performance improvement can be huge.

5.6.2 Applying the refined projected shadow texture

After uploading the refined projected shadow texture it has to be applied to the scene in another pass. This pass is the actual pass that renders to the screen.

This can be implemented quite straightforward. Basically, the pixel shader does not

anymore look up shadow information using the shadow map as described in section 5.1, but simply uses the information calculated by the CPU in the refined shadow texture, which has already been projected into eye view.

This method can introduce artifacts when using transparent objects due to the image-based nature, as objects behind the front transparent object look up the shadow information of the transparent object. However, these artifacts are hardly visible.

Chapter 6

Ray tracing

Ray tracing is a high-quality method for rendering scenes. Instead of rasterizing primitives like conventional 3D graphics cards, rays of light are traced throughout the scene.

The original idea of ray tracing is to create a large amount of rays beginning at a light source. Then each ray is checked for the closest intersection with an object. The material of this object then defines how to proceed, as the ray may be (probably perfectly) reflected and (partly or fully) absorbed. This may spawn one or several new rays, which are then recursively traced the same way.

While doing this, the ray may intersect the visible part of the camera's near plane. Once this happens, the remaining light color can be added to the screen buffer[1] at the appropriate location.

Of course, this algorithm is highly inefficient, as only a very small amount of light rays will finally contribute to the screen image, and many of these rays will only add a very small amount of light. An actual implementation of ray tracing therefore usually uses the opposite idea: tracing rays from the eye through the camera's near plane.

This chapter will give a short introduction to this algorithm. However, it will focus on shadow rays and performance improvements, as these have been the main research areas for the development of *City Viewer* to improve shadow quality in real-time.

[1] The screen buffer is initially filled with black

6.1 General algorithm

The actual ray tracing algorithm works differently than above. Instead of tracing rays from the light through the scene to the eye, rays are traced the opposite way[2]. Foley, van Dam, Feiner and Hughes explain that "ray tracing, also known as ray casting, determines the visibility of surfaces by tracing rays of light from the viewer's eye to the objects in the scene" [FvDFH96e].

This is done by selecting the eye as center of projection (and thus ray origin) and interpreting the screen as a window of a view plane. Usually the near view plane is chosen. The center of each 2D pixel on the screen is then transformed into the view plane. For each of those pixels, a ray is cast from the eye through the position of the pixel in the scene. In a naive implementation, each object is then checked for intersection with the ray in order to find the intersection closest to the eye.

Once the closest intersection has been found, reflection, refraction and shadow rays can be spawned. This thesis does not cover reflection and refraction rays, as those have not been needed for the application. Foley et al. [FvDFH96b] do provide a good introduction to them.

The shadow rays are spawned from the closest intersection to each light source in the scene. Again all objects are checked for intersection. If an object intersects a shadow ray, the current pixel is considered in shadow with respect to the light source[3].

6.2 Fast ray-triangle intersection

The rays need to be tested against different kinds of objects. While there are some object types (like spheres) that are very fast and easy to test against an intersection with a ray, the scenes *City Viewer* has to handle consist mainly of polygon surfaces which have been triangulated. Many algorithms exist for calculating ray-triangle intersections. However, many of those use very similar approaches with slight differences in detail.

[2] Actually, there indeed are ray tracing systems that trace from the light source. Those usually have special purposes and use involved optimizations to increase the amount of eye-hitting rays. They are usually called forward ray tracing systems, though some refer to them as backward ray tracing systems, as they use the opposite of the usual direction.

[3] If the intersecting object is semi-transparent, the shadow value must be dampened and the shadow ray tracing must continue from the point of intersection to the light source. *City Viewer* does not support semi-transparent object shadow casters currently.

City Viewer does ray-triangle intersections using a method described by Wald [Wal04]. The method (as most approaches) first calculates the signed distance t_{plane} along the ray to the plane specified by the triangle's vertices A, B and C. With the ray R specified by the origin O and direction D, this can be computed as

$$N = (B - A) \times (C - A) \tag{6.1}$$

$$t_{plane} = -\frac{(O - A) \cdot N}{D \cdot N} \tag{6.2}$$

t_{plane} is then tested against t_{min} and t_{max}, which specify the minimum[4] and maximum[5] signed distance allowed. If $t_{min} \leq t_{hit} \leq t_{max}$ is false, the ray intersection returns immediately. Otherwise, the hit point H is calculated by

$$H = O + t_{plane}D \tag{6.3}$$

Then the barycentric coordinates of H in the triangle can be calculated. Once the barycentric coordinates α, β and γ are known, they can be used to test if H is inside the triangle. This is true when the following condition is fulfilled:

$$(0 \leq \alpha \leq 1) \wedge (0 \leq \beta \leq 1) \wedge (0 \leq \gamma \leq 1) \tag{6.4}$$

Instead of testing $0 \leq \alpha \leq 1$, $\beta + \gamma \leq 1$ can be tested to avoid calculation of α. This can be done because of the way barycentric coordinates are defined ($\alpha + \beta + \gamma = 1$).

Although calculation of the barycentric coordinates can be done directly in 3D by solving a system of equations or using geometrical methods, Wald suggests to use the *projection method* described below.

6.2.1 Projection method

Wald explains that projecting both the triangle ABC and the hit-point H into any plane (except planes orthogonal to ABC) does not change the barycentric coordinates of H. By projecting both to one of the 2D coordinate planes, calculation of the barycentric coordinates can be done in 2D. Wald suggests to project to the plane with the maximum projected area, which can be found by comparing the absolute values of the components of N. The dimension with the highest absolute value in N is used as

[4] t_{min} specifies a small positive epsilon to avoid self-intersection of rays that have been spawned on surfaces

[5] t_{max} usually is the current shortest intersection distance found, if existing

"projection dimension". This dimension is ignored when calculating the barycentric coordinates.

Let A', B', C' and H' be the projected points of A, B, C and H.

$$H' = \alpha A' + \beta B' + \gamma C' \tag{6.5}$$

By substituting $\alpha = 1 - \beta - \gamma$, the term results in

$$\beta \left(B' - A'\right) + \gamma \left(C' - A'\right) = H' - A' \tag{6.6}$$

Wald shows that by solving this equation using the Horner scheme, β and γ can be calculated by

$$\beta = \frac{b_u h_v - b_v h_u}{b_u c_v - b_v c_u} \tag{6.7}$$

$$\gamma = \frac{c_v h_u - c_u h_v}{b_u c_v - b_v c_u}, \tag{6.8}$$

where $h = H' - A'$, $b = C' - A'$ and $c = B' - A'$. u and v specify the dimensions that are used for calculating the barycentric coordinates. For example, if the projection dimension is y, then x is used for u while z is used for v. As in implementations the different components of a vector usually are stored in an array, u and v can be calculated by the projection dimension k as follows:

$$u = (k+1)\, mod 3, v = (k+2)\, mod 3 \tag{6.9}$$

Instead of using the modulo operation, a lookup table can be used.

6.2.2 Precalculating the projection method

Wald points out that the projection method calculates many values each frame that are constant for a triangle. Like many intersection methods the triangle normal is used. The normal is constant and should be precomputed. To save memory, only the u and v components of the normal are stored. This is done by dividing N through N_k[6]:

$$N' = \frac{N}{N_k} \tag{6.10}$$

[6] N_k cannot be zero as k is the projection dimension

Thus N'_k is equal to one and doesn't need to be stored. This leads to

$$t = \frac{A \cdot N' - O_u \cdot N'u - O_v \cdot N'_v - O_k \cdot N'_k}{D_u \cdot N'_u + D_v \cdot N'_v - D_k \cdot N'_k} \tag{6.11}$$

$d = A \cdot N'$ is constant and can therefore be precalculated. It is stored along with N'_u and N'_v.

Similarly, calculation of the barycentric coordinates can be simplified, too:

$$\begin{aligned}
\beta &= \frac{b_u h_v - b_v h_u}{b_u c_v - b_v c_u} && (6.12)\\
&= \frac{1}{b_u c_v - b_v c_u}\left(b_u H'_v - b_u A_v - b_v H'_u + b_v A_u\right) && (6.13)\\
&= \frac{b_u}{b_u c_v - b_v c_u} H'_v + \frac{-b_v}{b_u c_v - b_v c_u} H'_u + \frac{b_v A_u - b_u A_v}{b_u c_v - b_v c_u} && (6.14)\\
&= K_{\beta v} H_v + K_{\beta u} H_u + K_{\beta d} && (6.15)
\end{aligned}$$

Only the constants $K_{\beta v}$, $K_{\beta u}$ and $K_{\beta d}$ have to be stored. The same precomputation can be done for γ.

6.2.3 Cache efficiency

As mentioned by Wald, preprocessing must be done carefully to avoid cache misses. A cache miss may easily cost more than doing the actual calculation on the fly. He proposes to use a triangle intersection acceleration structure for each triangle, which only holds the constants necessary for intersection. For optimal cache performance, these structures must be stored linearly in memory using a special *triAccel* array. That way, indices and vertices don't have to be touched when calculating intersections. This avoids many pointer indirections, which usually have a bad impact on caching. The structure must hold 10 values: 3 floats (d, N'_u, N'_v) storing the plane equation, 6 floats for storing the 2D line equations ($K_{\beta v}$, $K_{\beta u}$, $K_{\beta d}$, $K_{\gamma v}$, $K_{\gamma u}$ and $K_{\gamma d}$) and an integer for storing k. The values should be stored in the same order as they are used when intersecting. Also, considering the use of SSE as explained later, an alignment of 16 bytes for the plane equation (together with k) and each of the two line equations should be used to avoid costly unaligned loads.

The usage of a special structure for each triangle also allows to prefetch the next triangle while working on the current. Newer processors offer special prefetch instructions to do this.

6.3 Performance improvements

Up to now, only the basic ray tracing algorithm has been discussed, along with an efficient triangle-intersection technique. However, there are many other ways to improve ray tracing performance. Some limit the amount of intersections that have to be made, while others improve actual intersection speed using low-level improvements. Some of these techniques are presented here.

6.3.1 Space partitioning

This is the most obvious improvement that can be made. *City Viewer* already uses a kd-tree for culling invisible parts of the scene when rendering using the GPU. Basically the very same can be done to individual rays. If a ray doesn't intersect a kd-tree node, it won't intersect any child nodes or objects, too. For rendering using the GPU, occlusion queries are used to determine visibility of a node. For ray tracing, the node bounding boxes are instead tested for intersection.

However, it is not necessary to test all bounding box sides of a node due to the hierarchical structure of a kd-tree. Wald describes an algorithm that efficiently takes use of this hierarchy. He introduces a *current ray segment* $[t_{near}, t_{far}]$, which describes the parameter interval of the ray that intersects the current voxel. This segment is initialized to $[0, \infty)$ and clipped to the bounding box of the whole scene. During traversal this interval is updated incrementally as follows:

When traversing a node, the distance d to the node's clipping plane is calculated. d is then compared to t_{far} and t_{near}. If $d \geq t_{far}$ or $d \leq t_{near}$, the ray segment lies completely on one side of the splitting plane. This side will then be traversed, while the other side can be culled[7]. Otherwise, both sides must be traversed. The first one is assigned the ray segment $[t_{near}, d]$, and the second one $[d, t_{far}]$.

This algorithm has the advantage of only using 1D computations. It also doesn't calculate or store any overhead information like entry, exit or intersection points. As the tree is traversed in front-to-back order relative to the ray, this also allows to use *early ray termination*. Once a primitive hit is found after testing a leaf, the traversal can be stopped.

The algorithm also has another advantage: the space necessary for storing a kd node

[7] care must be taken to handle triangles that lie on the splitting plane, and rays parallel to the plane must be considered, too

is small, as it does not need to include information like bounding boxes. *City Viewer* doesn't use the original kd-tree structure for ray tracing. Instead, a cache optimized kd-tree as introduced by Wald is used for traversal. This tree is built using the original kd-tree. All nodes in the array are stored in an array, therefore taking better usage of the cache. Additionally, this allows to save memory for each node, as explained in figure 6.1. Each node stores a single bit inside an integer that specifies if it is

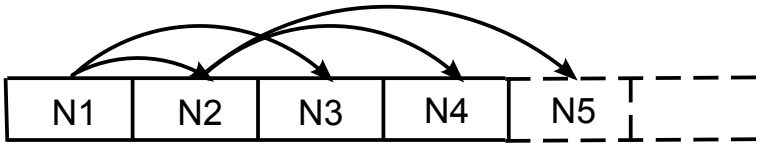


Figure 6.1: Memory layout of cache efficient kd tree

a leaf node. If it is, the other 31 bits of the integer specify the object count stored in the node. If the node is not a leaf, 2 bits of the integer are used for the splitting dimension, while the remaining 29 bits contain an offset to the first child of the node[8]. Another 32 bit union is used for leaves and parent nodes. Leaves contain a pointer to the objects inserted and parents contain the split value of the node. That way, a whole node only takes 8 bytes, thus allowing several nodes to be stored inside a single cache line.

6.3.2 Multithreading

Ray tracing is an algorithm that can easily be parallelized using multithreading. *City Viewer* does this by dividing the screen into quadratic bricks[9] of the same size. Each brick is associated with a flag that tells whether the brick has already been worked on. Several worker threads are started. Each one takes an unflagged brick, flags it and then applies the ray tracing algorithm to it. Once finished, it searches the next unflagged brick and continues. Testing and setting the flag is done using an atomic function that increases the flag integer and returns the new value. As the flag integer is initialized to 0, the worker thread only works on an brick if the value returned is exactly 1, not higher.

This brick approach using flag integers makes sure that each thread has approximately the same work. Fixed association of threads to bricks contains the risk of some threads having less work than others, especially as the ray tracer may just be refining parts of the image.

[8] as the node size is a multiple of 8, the lowest 2 bits can be assumed to be 0, which makes the offset effectively 31 bits wide

[9] A typical brick size is between 8x8 to 32x32 pixels

Other than the atomic operations for setting the brick flags, no synchronization is necessary, as only thread specific data is modified, while shared data is just read.

6.3.3 Caching shadow casters

The main reason for supporting ray tracing in *City Viewer* are shadows. Shadow rays do have an interesting property: it does not matter which object they hit, so finding the closest intersection is unessential. Instead, just knowing if there is an intersection or not does suffice[10]. This allows to use a cache list for each brick which contains the last primitives that have been intersected by shadow rays. Due to using one cache list per brick local coherency should be high, even between frames. Instead of immediately traversing the kd tree when testing a shadow ray, the cache of the current brick is searched.

6.3.4 SSE

Streaming SIMD Extensions (SSE) is an extension of the x86 architecture developed by Intel. It is an "**S**ingle **I**nstruction, **M**ultiple **D**ata" (SIMD) architecture, meaning the same instruction is applied to several data values. This is also called vector processor, as usually a vector of data is used for calculation. SSE uses 128 bit wide registers. As an example, instead of four 32-bit floating point multiplication instructions, a single instruction suffices. SSE provides several overloads for the specific operations to allow multiplication of two 64-bit floating point values just as four 32-bit values.

In our case, usually four 32-bit floating point values are handled at once. Although 64-bit precision may help with some precision issues, the significant performance advantages of using 32-bit precision values are preferred for the real-time rendering intentions of *City Viewer*.

The SSE version of the ray tracing algorithm traces four rays in parallel. Another option would be to trace one ray at once and test it against four triangles. However, the actual kd tree traversal, which takes a significant amount of processing time, is not parallelized that way.

[10] *City Viewer* does not support semi-transparent materials regarding shadows. Otherwise the closest intersection must be found for correct handling of semi-transparency

Data ordering

One problem with using SSE is data ordering. Usually, data is stored in an ***A**rray **o**f **S**tructures (AoS)* order. Rays, for example, are then stored like in figure 6.2. This

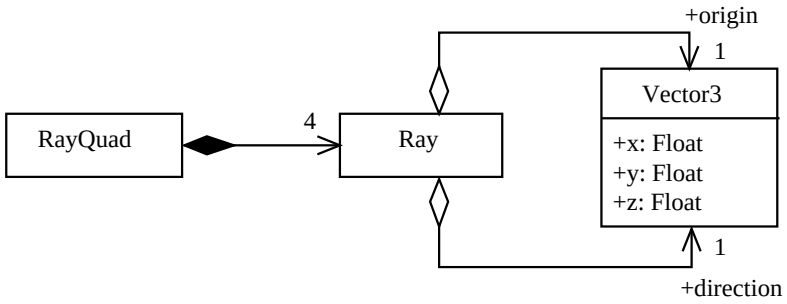


Figure 6.2: Array of Structure approach to ray quads. This is inefficient for SSE in practice, as parallelism is quite limited to a few instructions

does not lend well for using SSE instructions. Wald shows that an ***S**tructure **o**f **A**rrays (SoA)* order is much better. This can be explained by looking at figure 6.3. Each

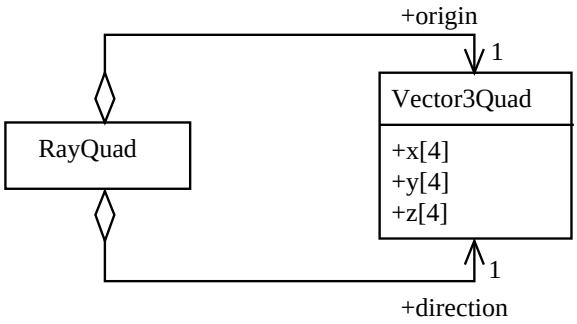


Figure 6.3: Structure of Array approach to ray quads. A ray quad can be handled exactly like a single ray in the non-SSE version.

line of code can be directly translated to just operate on four floating point values instead of a single floating point value, as each of those values has the same semantic meaning[11]. In contrast, the AoS approach would impose changing the actual algorithm to achieve good parallelism. In an AoS approach, a Vector3 could be accessed as SSE 128 bit value and the four floating point values would represent x, y, z and a padding, thus changing the semantic meaning of each single value. Although some parallelism could be introduced by using SSE operations for vector additions, subtractions and similar operations, it would be quite limited to specific low level parts. SoA however achieves parallelism for almost all operations.

[11] In this case, all four floating point values would for example represent x, y or z coordinates of four different vertices

Thanks to the SoA approach, SSE code is easy to apply to the given *Single Instruction, Single Data (SISD)* algorithms. To calculate the hit point on the uv plane, the following code is used in the non-SSE version:

```
float hu = rayOrigin[u] + d * rayDirection[u];
```

In the SSE version, four rays are processed in parallel. This can easily be written as

```
__m128 hu = _mm_add_ps(rayQuadOrigin[u], _mm_mul_ps(d, rayQuadDirection[u]));
```

Full parallelism can only be achieved using an SoA approach. It allows to directly convert each instruction to SSE. An instruction that adds the x components of two vectors then just adds the x components of two vector quads. Although some instructions can be parallelized without SoA (like vector adding), SoA does not offer full parallelism for more complex algorithms (like matrix-vector multiplication). Klimovitski [Kli01] offers a more detailed introduction to SSE along with efficient ways of converting AoS structures to SoA, as naive conversion algorithms are subject to memory stalls.

SIMD traversal

Traversal of the kd tree using SIMD on four rays is basically done like traversal of a single ray. Four rays[12] are put into a ray quad, which is then processed at once. All four rays of the quad are put together and follow the same traversal path until all rays have finished. Some special cases have to be handled though:

- Not all rays may need to visit a certain node. However, if just one ray needs to visit the node, the whole quad must visit the node. This is rarely the case, as usually ray coherency is strong.

- This also implies that the whole quad must be traversed until all rays have hit or the traversal stack is empty.

- The front-to-back traversal order for the individual rays may differ. Wald shows that this can be avoided by using rays that either have the same origin or the same direction signs, as those always have the same front-to-back traversal order. *City Viewer* uses rays that have the same direction signs, as those can be handled less costly and fit nicely to shadow rays.

[12] in the actual implementation sometimes less than four rays may be in a ray quad

Implementation caveats

Although most parts of the traversal and intersection code can be directly converted to SIMD code, some issues have to be taken care of. First of, branching should be avoided because of the deep SSE pipelines. Every branch can cause a serious stall. Therefore branching should be replaced by conditional moves, although this may lead to executing more code than when using a branch.

When traversing the kd tree, some rays may intersect the split plane while others do not. Still the same operations will be applied to each of the rays, which makes it necessary to deactivate the rays not intersecting the node. Wald suggests to do this by setting the ray segments of the near side to $[t_{min}, min\,(d, t_{far})]$ and the segments of the far side to $[max\,(d, t_{near})\,, t_{far}]$. This avoids that the segments of non-intersecting rays get longer than before. By above min/max operations, rays that are invalid for the current node have negative segment length, which allows to use SSE instructions to generate a bit mask. This bit mask can then mask out invalid rays with no need for conditionals.

Chapter 7

Application design

Development of an application is subject to many pitfalls. The later these become known in the development process, the harder it is to take care of them. Considering this, it is important to design the application carefully before doing any implementation work. This chapter gives some insight into the general design of *City Viewer* along with measurements of application performance depending on the shadow technique being employed.

7.1 Packages

Reusability of code has been perceived as important while developing *City Viewer*. Parts of the code will probably be reused when implementing similar applications like a CityGML editor. Therefore CityGML code should be completely separated from actual viewer code. Additionally a DirectX 10 application framework shall ease development of future applications. Figure 7.1 shows all important packages used for *City Viewer*.

The implementation of the *City Viewer* application consists of the following component packages:

Console Provides a console interface for debugging purposes

CityGML Handles interpretation of the *CityGML* XML files. XML parsing is done using libxml.

libxml2 A XML parser library available under the MIT license.

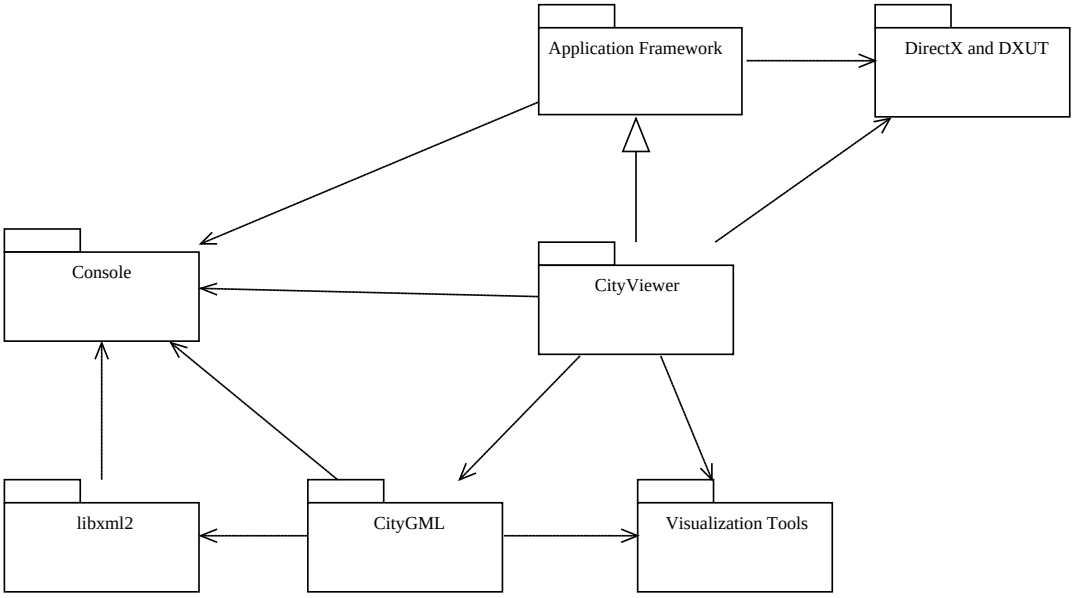


Figure 7.1: Package overview of the City Viewer *application*

Application framework Provides a basic framework for DirectX applications, based on the D3DX library. This includes dialog and document/view support along with helper classes for several Direct3D 10 interfaces.

Visualization Tools Provides a math library for matrices and vectors along with several other classes like hyperrectangles, KD trees and triangle mesh splitting.

City Viewer This component is responsible for user input and displaying the GML file using Direct3D 10.

7.2 Console

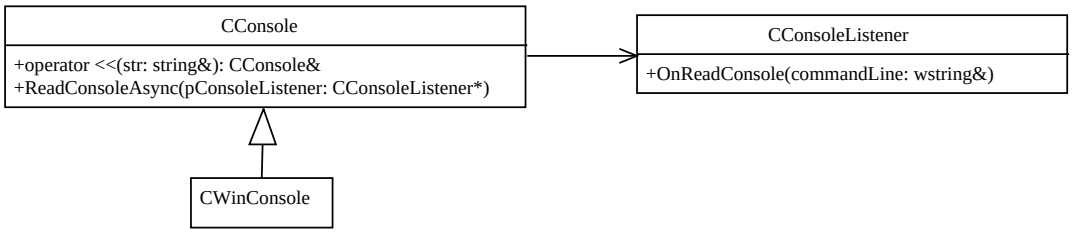


Figure 7.2: Console package

This package provides access to a console system. Although consoles aren't anymore a good choice for user interaction, they can still provide valuable debug information for development.

The abstract *CConsole* interface is shared by all implementations. Additionally to screen output, *CConsole* implementations also support writing to a log file. Currently there is only a Microsoft Windows console implementation which uses the Win32 API for direct console access. Other possible implementations are graphical consoles (using DirectX, OpenGL or window systems) or using STL for accessing the standard input and output streams. The package also offers macros for quick display of color coded error, warning and debug messages and is therefore used by almost all other packages.

7.3 Application framework

When developing applications of a specific domain, often the very same work has to be done for every application. Thus development speed can be increased easily by providing a framework that performs these tedious tasks automatically. Some of the more general tasks when developing applications are:

Event handling Provide a general event handling that forwards messages to the appropriate handlers (like views or buttons)

GUI Creating a main window and providing dialogs and controls

Document-View pattern Providing base classes for the Document-View architectural pattern (or Model-View-Controller pattern)

Command model Base classes for commands to allow undo/redo functionality

Domain specific support Functionality for common tasks in the given application domain

In the context of DirectX graphics applications, domain specific support can be categorized as follows:

Encapsulation Encapsulate certain DirectX interfaces to ease usage

Resource loading Provide easy loading of resources like effects or textures

Rendering Extend the Document-View model by making sure views support several rendering passes

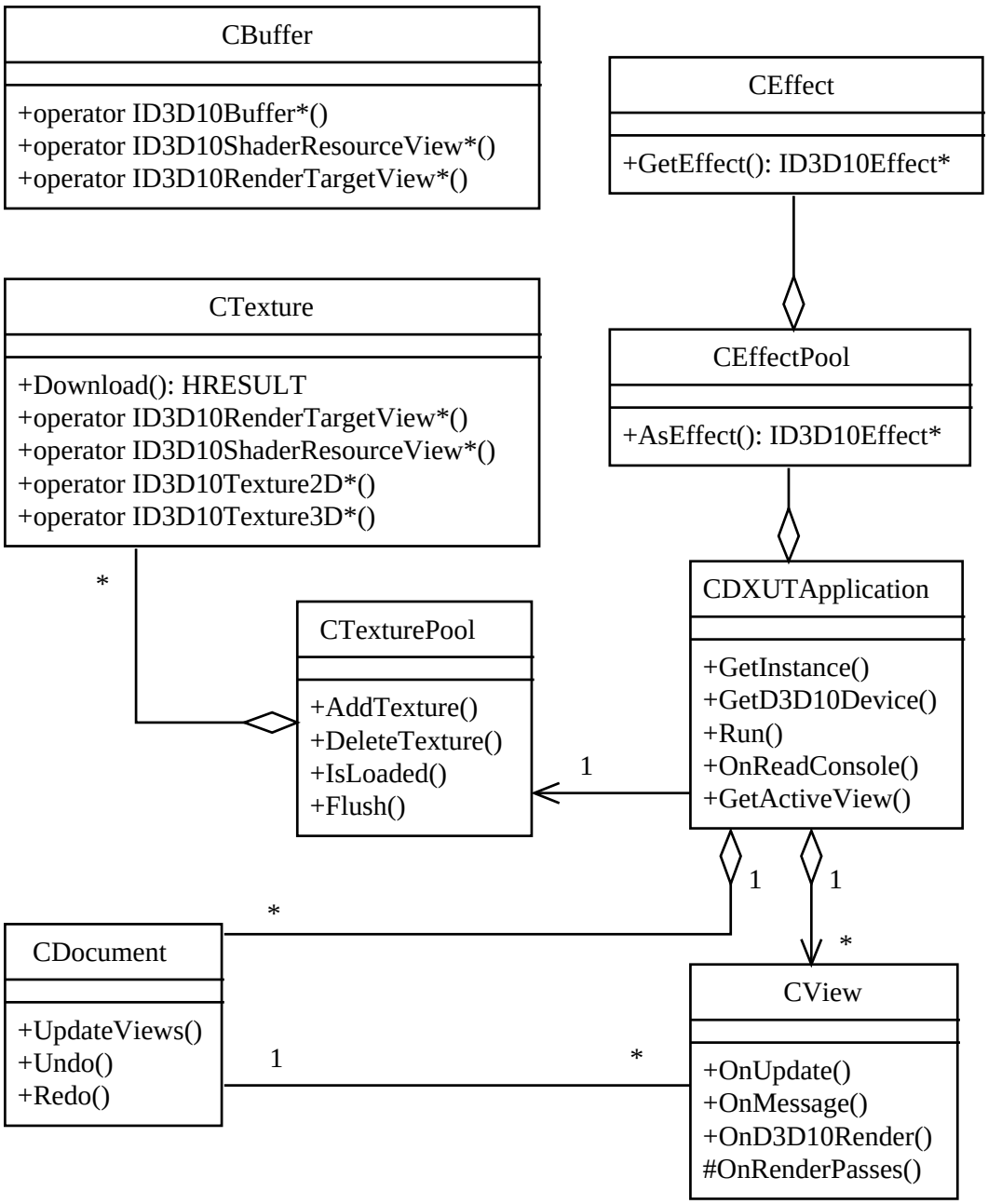


Figure 7.3: Application framework for DirectX 10 applications

7.3.1 GUI

Although using a more general GUI library has been considered, the GUI is based on the *DirectX Utility Library (DXUT)* shipped with the DirectX SDK, which already offers a quite complete implementation of dialogs and basic controls. DXUT displays controls directly using the DirectX API, instead of relying on windows controls.

The application framework provides extensions to the dialog model of DXUT. It adds

basic docking functionality for dialogs, so that the user can attach dialogs to each side of the screen. Together with minimizing dialogs (support included in DXUT), this allows cleaning up the screen for complex applications.

7.3.2 Textures and effects

3D graphics applications need access to textures and shaders. Therefore the application framework offers convenience classes for textures and effects. That way some code redundancy can be avoided, as quite often the same initialization work has to be done.

Pooling

Shaders and texture are needed at several locations of a complex application. In order to save memory, no file texture should be loaded twice. Also, compiling complex shaders can be quite time consuming, especially if they use loop unrolling, meaning that compiling a shader twice should be avoided. For these reasons a central pool is offered to allow easy access to textures and shaders.

Textures

The framework offers several texture classed (all derived by *CTexture*) for the different texture types, e.g. *CTexture2D*. Each of these classes supports creating textures for different purposes like render targets or depth stencil buffers. The ease of use comes with the loss of flexibility. However, the framework also supports using DirectX 10 interfaces directly with full flexibility, therefore this is a small price to pay.

The texture pool references textures just by filename. If a component requests a texture, the texture pool checks if the texture has been loaded. If so, the texture is directly returned, otherwise the pool tries to load the texture and return it. A possible extension is preloading the textures in another thread and only blocking if the texture is actually needed for display.

Effects

Effect files and subsequently shaders are treated differently to textures. In DirectX 10, shaders are usually accessed by effect files, which can contain several techniques consisting of several passes. Each pass can use completely different vertex, geometry and pixel shaders along with rendering states.

Effects can have child effects. The parent effects are called effect pools. This aims at limiting state changes and reusing shader constants, which can be shared across different effects. The framework supports this by introducing an effect pool class additionally to normal effects. The *CEffectPool* may have an associated DirectX effect pool, but this is not mandatory.

Effects and effect pools are loaded when initializing the application. The developer specifies a XML file that contains the filenames of all effect pools and effects. This XML file also offers some abstraction from the actual effect code by providing variable name translation. It also supports "#define" directives if needed for the actual effect shader code.

7.3.3 Document-view model

The framework provides support for a document-view architecture pattern. Although a model-view-controller pattern could be used, too, for graphics applications view and controller often can be merged into a single class.

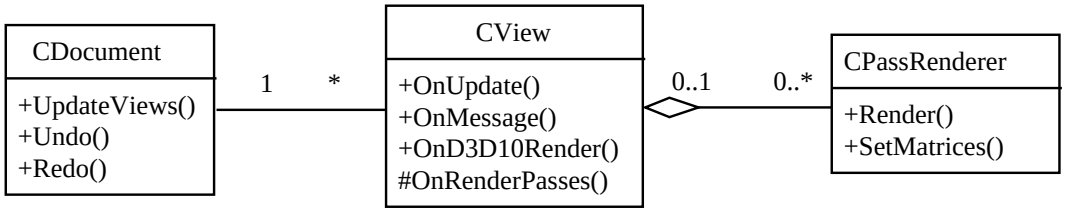


Figure 7.4: Document-View architecture

In this framework a view usually renders a given document using DirectX. However, complex rendering usually consists of several passes. For example, a shadow map can be generated and applied to a scene, which results in two passes. Also, those passes may be required for different types of views. This lead to the introduction of *CPassRenderer*, which does the actual rendering of a given scene. This concept is illustrated in figure 7.4.

7.4 CityGML

As has been shown in chapter 2, CityGML is a XML language. It describes a class hierarchy, which can be represented as C++ class hierarchy. The actual implementation directly translates the given hierarchy to a C++ class hierarchy. This eases following the (City)GML standard and allows for updating the library to newer revisions of the standard. An excerpt of this C++ class hierarchy including embedding classes is shown in 7.5.

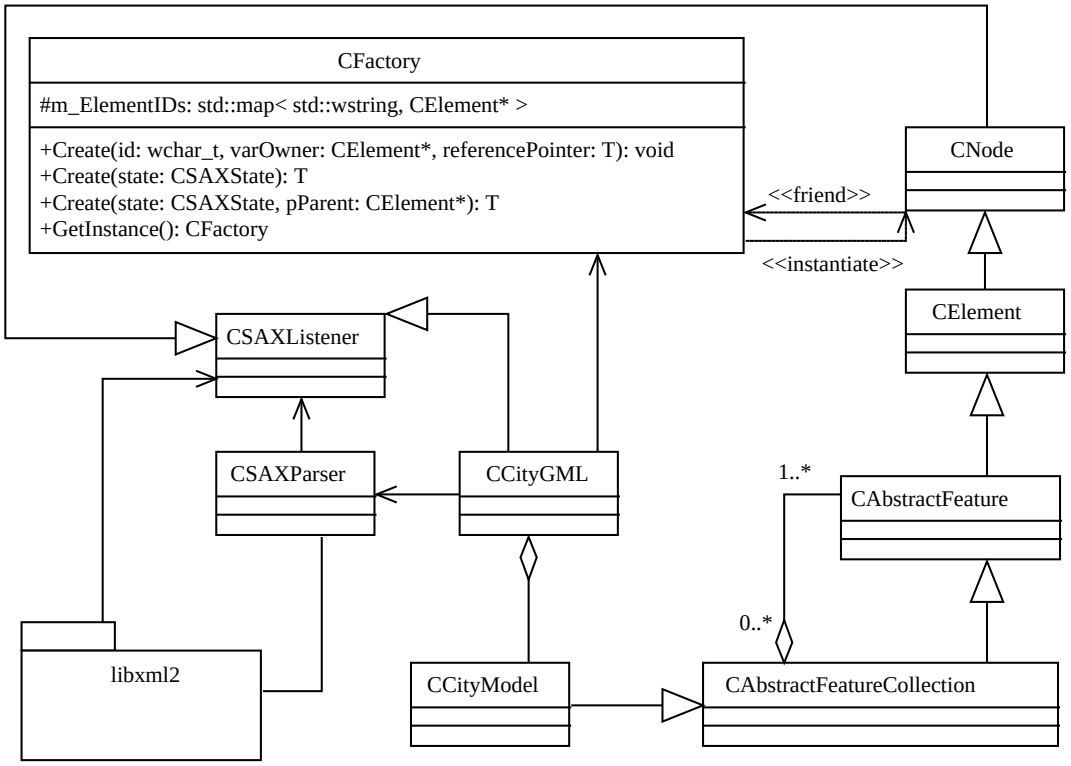


Figure 7.5: Excerpt of CityGML parser design

7.4.1 Graph construction

CityGML documents support linking of elements to each other. This is used extensively in the appearance model, as can be seen in figure 7.6. CityGML documents are thus no trees but graphs. As a consequence, object instances can be linked by several other objects. As C++ does not have garbage collection, a method must be found to ensure that objects are only destroyed when all their referencing objects have been destroyed. This is done similarly to the *Component Object Model (COM)* by adding *AddRef()* and *Release()* methods to *CNode* (which is the base class for all XML nodes). When constructing an element, its reference count is set to one. Every time an object

reference is stored, *AddRef()* must be called for this object, increasing the reference count by one. Once the reference is not needed anymore, *Release()* must be called to decrease the reference count. If the reference count reaches zero, the object is finally destroyed.

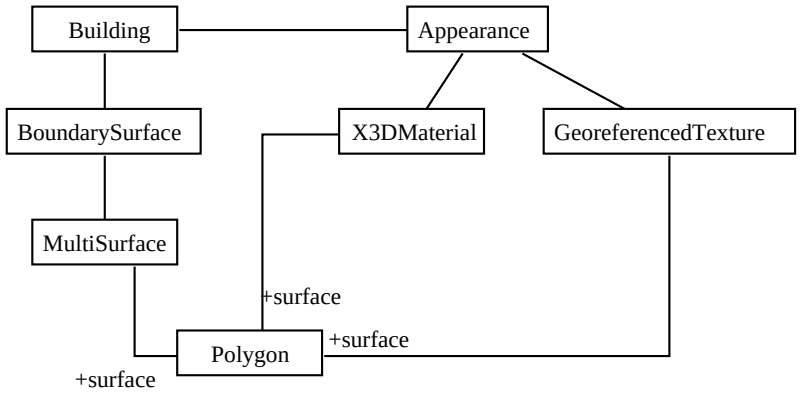


Figure 7.6: Object diagram of typical appearance model

Linking in GML is done using IDs. Each object that can be linked does have an ID. In order to support linking, a central factory class is introduced. This class is responsible for creating any C++ XML object. If the object does have an ID, the object and ID is stored in a map. The same factory class also provides a method to retrieve an object by providing an ID. The latter is complicated by the issue of SAX parsing, which is explained later. During SAX parsing, the objects are created linearly in the same order as they appear in the file. However, as an object in a XML file may be linked to before instancing it, those links cannot be directly established. If that is the case, the factory provides a callback mechanism to update the linking object once the linked object has been created.

7.4.2 XML parsing

Parsing the XML files is done using libxml2, which provides two different interfaces for reading:

DOM The *document object model* is the easiest way to access XML data. The given XML file is represented by a tree which can be traversed. This tree is fully in memory.

SAX The *Simple API for XML* uses another approach for accessing the XML data. Instead of parsing the whole file and providing access to the data afterward, *SAX* parses single elements of the file and directly sends them to the application, which is responsible for handling or storing this data in an appropriate way.

Both methods were considered for *City Viewer*. While the *DOM* approach was easy to implement, it was causing large amounts of memory to be wasted, as there are essentially two trees at once in memory: the XML parser tree and the *City Viewer* specific CityGML tree. Also the XML tree stores all data as string, and depending on string pool support of the parser, even the type of each node as string. Loading a XML file resulted in a memory usage larger than the stored file, as each node holds information about parent and other tree information along with the actual data. This has been considered a waste of memory, therefore the *DOM* approach has been skipped and *SAX* parsing was employed.

SAX parsing with libxml2

However, *SAX* parsing is slightly more complicated. The library *libxml2* offers SAX parsing by callback functions. The class *CSAXParser* provides these callback functions and forwards them to the current C++ object being parsed. The latter is implemented using a listener model. The listeners are stored on a stack, as after the object has been finished, the old listener must receive notifications again.

Once a new XML element is parsed, the current listener is notified and receives information about the element. The listener can then either skip the element[1], create a new C++ object representing the element[2] or use a state machine for handling the element. The state machine is often used for handling elements that further specify their included element. An example for this in CityGML are <lodXGeometry> elements (with X being a number), which specify the level of detail of their child element, which in this case must be a subclass of *_AbstractGeometry*. The state machine avoids having to create own C++ classes for elements like <lodXGeometry>, which contain no useful data themselves, but just add meaning to their embedded children.

Figure 7.7 shows a sequence diagram of a typical SAX parsing operation. Initially the current listener is *Building*. An element <boundedBy> is parsed, which signals that a element derived by *_BoundarySurface* will follow. This information is stored as current state of *Building*. *Building* now receives another *OnStartElement()* message with the type of boundary surface. It uses this information to create the appropriate WallSurface. The WallSurface registers itself as now listener and receives all child elements of the <WallSurface> element. Once all child elements have been parsed, the SAX parser removes *WallSurface* from the listener stack and notifies *Building* that the WallSurface has finished.

[1] including all children
[2] this causes the new object to be the new listener

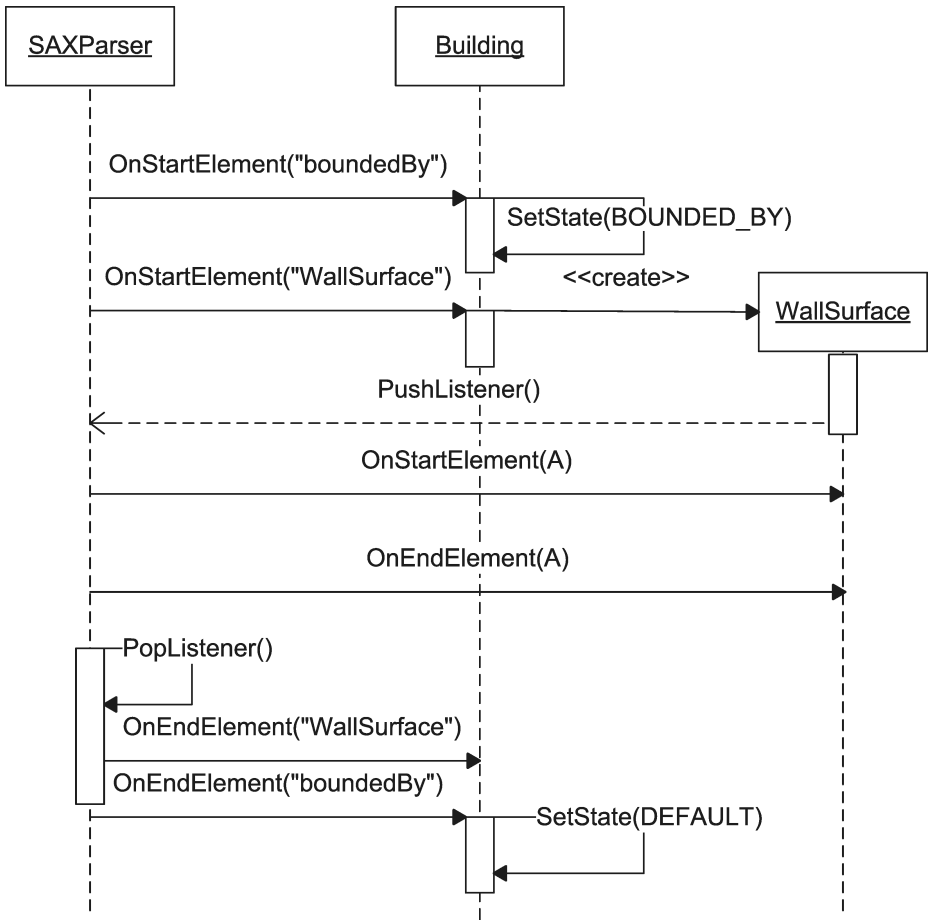


Figure 7.7: SAX parsing sequence diagram

7.5 City Viewer

The actual main application package is called *City Viewer*. It is based on the application framework and uses the CityGML parser package for reading CityGML files. The main package is responsible for integrating the different algorithms explained in this thesis along with implementing the actual display algorithms. A general overview of the package is given in figure 7.8.

The application's entry point is given by $CCityViewer$, which is responsible for things like for creating the application window, initializing Direct3D and the console system, loading effect files or creating the user dialogs. The user dialogs and controls are provided by the application framework, which bases on DXUT.

The document-view architecture is implemented by $CScene$ and $CSceneView$ (including child classes), which are explained later. Although the framework supports multiple documents, *City Viewer*'s architecture is limited to one scene for performance reasons. However, support for composing a scene using several sources (not limited to

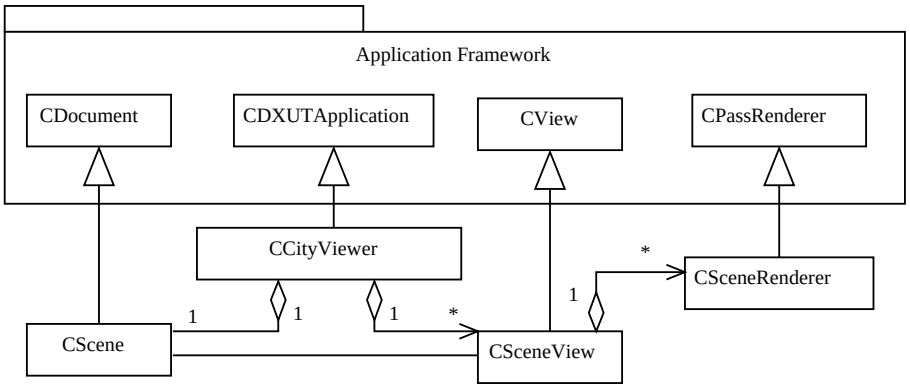


Figure 7.8: Excerpt of the City Viewer class model

CityGML) can easily be added.

7.5.1 Scene

The actual scene data is stored in *CScene*, which is derived from the application framework's *CDocument*. The scene is storing a *CEntity* collection. Each entity is representing a scene object, like a house or river. An entity may consist of several *CGeometry* objects. This concept is shown in figure 7.9.

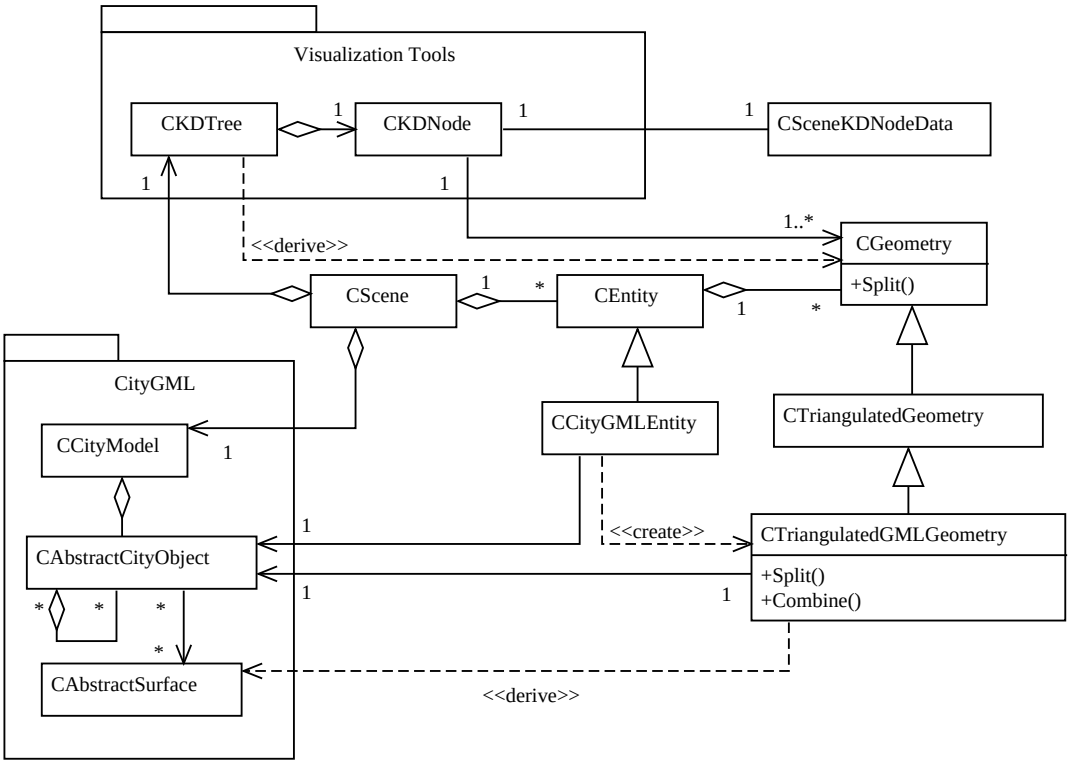


Figure 7.9: City Viewer*'s scene model*

The CityGML file is loaded using the CityGML package. Afterward, a collection of *CCityGMLEntity* objects is created, each one based on one *CCityModel* child. The *CCityGMLEntity* object then requests all (leaf) GML surfaces attached to the entity and builds one *CTriangulatedGMLGeometry* object for each surface. During this process the GML surfaces are triangulated and the results stored in each geometry object. *City Viewer* optionally supports to attach default textures and materials to the surfaces, based on the type of feature owning them, which is the first *CAbstractFeature* in the path from the surface to scene graph root.

At this stage, the very top and bottom of a city object are referenced in the CityGML scene graph[3]. This results in quite many *CTriangulatedGMLGeometry* instances, each probably only including very few triangles. As *City Viewer* is no geometry editor and picking shall only be done for CityGML features (not single geometries), this doesn't make sense. Therefore, *CTriangulatedGMLGeometry* objects are combined if they share the same theme and are owned by the same *CAbstractFeature*. A reference to the owning *CAbstractFeature* is kept. That way, algorithms can associate all CityGML features in the scene graph with their corresponding scene entities and geometries.

After combining geometries, vertices are shared. Sharing performance is increased by sorting the vertices along a coordinate axis. Changing the vertex order may decrease rendering performance, as described in chapter 4.4. Due to the way of converting each geometry object to a draw batch, which is rendered at once, this usually can be neglected. Additionally, the vertex buffer could be optimized before uploading to the GPU to avoid extreme cases. After sharing, vertex normals can be calculated[4].

Once all entities including their geometries have been built, the geometries are inserted into a kd-tree as specified in chapter 3.2. This kd-tree is then used for creating the draw batches as explained in chapter 4.4.1. Each kd-tree node may be associated with an application specific data object. In this case, it is associated with *CSceneKDNodeData*, which holds information like hyperrectangles or information needed by occlusion culling. This has been used to distinguish between optional data and data characteristic for each kd-tree. A side effect is much better cache usage when simply traversing the tree without accessing optional data, as optional data needs much more memory than pure nodes. This has been important for the ray tracer[5].

A kd-tree used for occlusion culling should be much less detailed than a kd-tree used

[3] Ignoring rings and points, which are even deeper in the scene graph

[4] Objects with materials that need face normals are treated when creating the buffer collections for the GPU

[5] Initially, the ray tracer used the original kd-tree. It now uses a special flat kd-tree optimized for cache compliance

for raytracing. This is due to the latency of occlusion culling and the pure processing power of the GPU. To avoid building two trees, all leafs and all nodes that exceed a certain depth are simply marked as leafs for GPU rendering. Finally, all draw batches are created at these nodes and the resulting buffer collections are uploaded.

Temporarily, this process leads to redundantly stored geometry. Geometry is stored with different compression methods at several locations:

CityGML The CityGML class hierarchy stores plain vertex positions and optionally texture coordinates. Data is stored as uncompressed 64-bit floats

CityViewer Vertex positions, normals and texture coordinates along with indices are stored. Vertex positions are shifted towards the scene center and stored as 32-bit floats. Vertex sharing is performed. Optionally, an acceleration structure is stored for each triangle to increase raytracing performance

Buffer collections All vertex and index data is stored compressed. The compressed data is temporarily stored in main memory before uploading it to the GPU

This data layout can be problematic for large scenes, especially since the CityGML scene graph itself has considerable memory cost even without the actual vertex data. This is also true for the CityViewer layer when creating deep kd-trees, as each *CGeometry* object adds overhead. Deep kd-trees decrease the triangle count of single *CGeometry* objects while increasing the count of objects.

However, it is possible to decrease the memory load. First, the scene graph geometry information of CityGML can be removed after constructing the scene. This leads to either not supporting (geometry) editing functionality or at least to losing the original geometry hierarchy when storing the scene to a GML file. Second, the acceleration triangle structure can be avoided, resulting in reduced raytracing speed. Finally, after uploading the geometry data to the GPU, no geometry data has to be stored in main memory at all. However, this disables (CPU) raytracing and picking.

7.5.2 Views and renderers

City Viewer supports several different views. Each view is responsible for displaying the scene a certain way to the user, while a renderer represents just one pass of doing that. The view and renderer class hierarchy is shown in figure 7.10

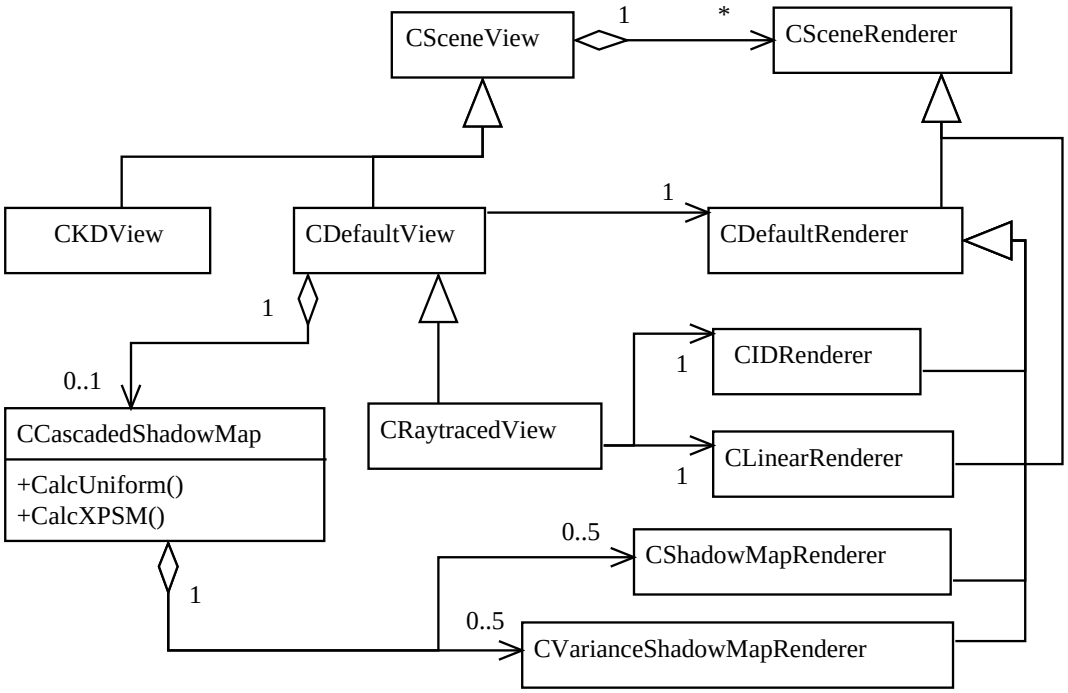


Figure 7.10: City Viewer*'s view and renderer model*

Renderers

There are several types of renderers. Basically two types of renderers can be distinguished: those using occlusion culling and those rendering batches linearly. All renderers that make use of occlusion culling are derived by $CDefaultRenderer$, which implements the algorithm described in chapter 4.6. The most important renderers are:

CDefaultRenderer Implements the occlusion culling algorithm along with rendering batches using their associated appearances. Supports back-to-front rendering of transparent objects and applying a depth or variance shadow map. Also stores a list of leaf nodes that passed occlusion culling

CIDRenderer Renders to a 8-bit integer RGBA texture and stores an unique primitive ID in the first three components. The alpha channel is used for flagging shadow and reflection raytracing information. Uses occlusion culling

CShadowMapRenderer and CVarianceShadowMapRenderer Renders a depth or variance shadow map. Ignores transparent objects and uses occlusion culling

CLinearRenderer Renders the draw batches of the given leaf nodes using their appearance. Also applies a prerendered screen-space shadow mask instead of a

shadow map

CCascadedShadowMap Although not directly a renderer, serves a similar purpose. Encapsulates up to 5 depth or variance shadow maps as presented in chapter 5.5. For each shadow map a separate renderer is allocated, which renders into one texture of the texture array. Uses visibility information given by the occlusion culling tests of the last frame's camera renderer to further narrow down the focus area of each shadow map[6]

Views

There are currently three different types of views:

CDefaultView Renders the scene using $CDefaultRenderer$. Before rendering the scene on the screen, a cascaded shadow map can be generated by *CCascaded-ShadowMap*.

CRaytracedView This view consists of several passes. First, the cascaded shadow map is rendered as usual, forcing variance shadow maps[7]. Then, an ID renderer is used to create an ID texture with shadow edge information. A ray tracer is invoked to create the refined screen-space shadow texture. Finally the scene is rendered to the screen using $CLinearRenderer$, which applies the refined texture.

CKDView Simply renders the kd-tree including objects in wire frame mode

Both $CDefaultView$ and $CRaytracedView$ support screen-space Gauss filtering for the shadows. This is done using a renderer similar to $CIDRenderer$, which just outputs a shadow image in screen-space. The gauss filter is then applied to the shadow image. Afterward, $CLinearRenderer$ is used together with the smoothed shadow image to render the scene.

Picking $CDefaultView$ and all descendants support picking, which is simply implemented by invoking the ray tracer. The pick returns a $CGeometry$ instance, which keeps a reference to the deepest CityGML $CAbstractFeature$. This feature is then

[6] With very low frame rates and while moving the camera rapidly, this may result in shadows missing for scene parts coming into view. Usually this is not visible to the user as most times it happens at areas far away

[7] The shadow maps are focused on the area visible last frame, which is given by the ID renderer

selected. By scrolling up and down with the mouse wheel, the user can select the different CityGML parent features. Rendering is done by marking all draw batches that contain some selected geometries. These draw batches are then broken up during rendering to distinguish selected geometries from those that are not selected.

Information text about the currently selected object is shown on screen, centered on the object. If the object is currently not visible in view frustum, the text is shown at the screen side closest to the object.

7.6 Performance

Table 7.1 gives a short overview of the achieved frame rates depending on the shadow map technique being employed. The frame rates just give a rough estimate of relative performance, as each of the techniques is still subject to ongoing tuning. Especially the raytracing approach can be tuned by optimizing the kd-tree and improving edge detection. The latter can also be used as a trade-off between performance and quality.

	No smoothing	Gauss smoothing ($\sigma = 20$)
Depth shadow map	47.7	18.59
Hybrid	9.34	6.85
Variance shadow map	19.87	-

Table 7.1: Frame rates measured for different shadow variants. All measurements have been taken at a screen resolution of 800 × 600 with 3 cascaded XPSM shadow maps sized 1024 × 1024. Rendering has been done on a Pentium Core 2 Duo 2.2GHz with 2GB RAM using a NVIDIA GeForce 8700M GT (512 MB). The variance shadow map is smoothed by a 5 × 5 Gauss kernel when sampling

Figures 7.11 to 7.15 show parts of the scenes as rendered for table 7.1. They show a CityGML test dataset that has been created based on the city of Ettenheim, which is available on the CityGML website.

Figure 7.11: Shadow map without smoothing

Figure 7.12: Shadow map with strong smoothing

Figure 7.13: Raytracing hybrid without smoothing

Figure 7.14: Raytracing hybrid with strong smoothing

Figure 7.15: Variance shadow map

Chapter 8

Conclusion and prospects

This thesis has presented an extensible city model viewer with respect to high-quality shadows. CityGML has been chosen as media type due to good documentation and promising prospects. CityGML files are a XML language and therefore can be parsed by any XML parser. To avoid memory issues caused by DOM parsers, an SAX approach has been taken. The CityGML class hierarchy has been directly translated to a C++ class library, which is used by the main application for loading CityGML files.

The CityGML hierarchy is transferred into a flat structure with triangulated surfaces. This structure is then inserted into a kd-tree using a surface area heuristic. This heuristic has proved to result in very efficient kd-trees and can easily be adapted to support time as dimension. However, construction of the kd-tree using the surface area heuristic is far more expensive than simpler construction proceedings.

Rendering performance can be vastly improved by using draw batches to avoid state changes and limit draw calls. Additionally, the kd-tree lends itself well for hierarchical occlusion culling. This improves performance significantly, as CPU stalls are avoided by hiding the latency.

Shadows have been considered an important part of the thesis, as they are necessary for creating realistic images. Several real-time shadow techniques have been compared. For the purposes of this thesis, shadow maps have been chosen for providing the best compromise of speed and quality. It has been shown that shadow map aliasing can be reduced using several techniques. While perspective shadow map approaches offer decreased aliasing for almost no performance cost, some of them are subject to several implementation caveats. Cascaded shadow maps are more expensive due to rendering the scene several times, but they are easier to implement and offer better quality. However, none of these techniques is capable to fully hide aliasing artifacts. The

artifacts can be further reduced by using variance shadow maps or screen-space Gauss smoothing.

To produce (almost) perfect shadows, shadow volumes or raytracing can be used. Shadow volumes are well known, but have not been considered for this thesis, as they don't scale well with large (and detailed) scenes. Instead, an hybrid approach of shadow maps and raytracing has been examined. It has been shown that it is possible to refine shadow edges using raytracing interactively. Although this is still quite CPU intensive, raytracing scales very well with increasing parallelism. As the processor core count is expected to rise further, this method may become interesting for applications with high performance requirements, for example games.

The following sections conclude this thesis with proposals for further research to improve the shown techniques.

8.1 Caching

As creating high-quality kd-trees is expensive, they can be cached on the hard disk when initially loading a city. However, simply loading the CityGML file to memory can take some time due to large file sizes. This would still be necessary for querying the checksum to make sure the kd-tree is up-to-date.

Another option is to completely avoid loading the CityGML file after the cache file has been created. The cache file doesn't have to store all data available in the CityGML file. Information can be accessed by storing file offsets into the original CityGML file and load entity data stored at that location. To support elements linked by that entity, for each element ID an offset should be stored in a global list.

8.2 Tiling

Although the viewer has been designed to deal with quite large scenes, limits are reached easily. High-detailed cities of several square kilometers quickly reach memory limits of both graphics card and main memory. This can be solved by tiling a city into several smaller parts that are loaded on demand. Any solution has to deal with the difficulty of main memory limits. This section proposes an idea for creating the tiles.

The SAX parser is invoked on the CityGML file[1]. A CityGML file includes the city's extent at the beginning of the file. The number of tiles can then be calculated heuristically by file size and city extent. The entities traversed are associated to the appropriate tiles and a kd-tree is built for each tile. As only a limited amount of tiles can be stored in memory (as the algorithm is assumed to be running on the same machine as the viewer), the tiles may have to be swapped to hard disk. This is expensive and complicated to implement.

An easier approach is to use multiple passes. Each pass is responsible for a certain amount of tiles[2]. An entity that doesn't belong to the currently active tiles is ignored. Otherwise, the entity is parsed as usual and added to the current tile. Once all entities of the current pass have been parsed, the kd-tree is built and stored on hard disk. Then, the SAX parser is invoked again and the next tiles are handled. This is continued until all tiles have been created.

The actual viewer then has to load the cached tiles instead of the CityGML file. It must be made sure that preloading tiles when moving the camera is done smoothly. This can be done using several techniques, like considering the current camera movement.

8.3 Tuning

Generally, large parts of the thesis can be tuned to a large degree. This ranges from kd-tree building constants to choosing the best count and resolution of cascaded shadow maps. Shadows in general offer a wide range of options, from using raytracing to Gauss smoothing or variance shadow maps. Each of those techniques has a set of options to tune, with some of them influencing other parts of the shadow process.

8.4 Functionality

Current user functionality of *City Viewer* includes viewing a scene, moving through it and being able to pick CityGML elements. When picking an element, the type of element along with any string attributes is shown on the screen. This functionality can be extended to include any information available for the feature. Most of this data is currently ignored when loading the scene. Adding parsing and display code for that

[1] It could also be invoked on a number of CityGML files

[2] It's not necessary to limit each pass to only one tile, as tiles must be small enough to store some of them at once on the graphics card when rendering

kind of information is straightforward.

It may also be interesting to add editing functionality to the viewer. While providing geometry editing seems to be out of scope, it makes sense to add feature editing, including transforming features, changing feature data (especially time information) and adding or removing features. The implementation would have to deal with kd-tree issues, as fully rebuilding the kd-tree each time an object is moved is prohibitive. However, moved objects don't have to be stored in the kd-tree, or they may simple be inserted into the tree without changing the tree structure[3].

Adding animated data would be another step to higher realism. Depending on the type of animation, the animated features are subject to the same kd-tree issues faced by edited features. Some solutions to this have been proposed in chapter 3.2.

8.5 Ray tracing

The current implementation splits polygons that intersect a kd-tree node. This is ideal for the occlusion culling algorithm employed. However, polygon count increases drastically the further subdivision continues. Occlusion culling doesn't need deep subdivision to work well, but raytracing performance is very much dependent on the depth of the kd-tree.

To make deep kd-trees more efficient, triangles should not be split anymore after reaching the rendering leaf layer. Instead, references to the polygons should be kept. The ray tracer can use a technique called mailboxing to avoid intersecting a primitive several times [WBWS01]. Each ray (or ray quad) is assigned an unique ID, and each primitive stores the ID of the last intersected ray. An intersection can be avoided if the IDs match, as the results would be the same anyway.

Another idea is to use GPU ray tracing to refine shadows, which is currently investigated by the chair for Computer Graphics and Visualization at the Munich University of Technology. An advantage of that approach is given by the fact that CPU ray tracing of shadows leads to full usage of the CPU, although it can be done interactively by masking out parts of the screen. However, real-time applications often need to do other work than just visualization. As an example, games need to do expensive calculations for collision detection and artificial intelligence. This would be significantly more difficult to implement efficiently. A GPU ray tracer instead allows to do other calculations on the CPU. Especially for real-time applications like games, GPU ray

[3]This limits kd-tree efficiency, of course

tracing of shadows therefore may be much more interesting.

Ray tracing has a serious advantage: it behaves logarithmically in regard to scene complexity, while conventional rasterizing methods behave linearly. Combined with the fact that scenes are becoming more and more complex, ray tracing seems to be the “better” algorithm in a few years. This thesis has shown that it already can be used today for high-quality shadows in hybrid approaches.

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franọçois Yergeau (editors). Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/2006/REC-xml-20060816`, 2006.

[BS02] William Bilodeau and Michael Songy. Method for rendering shadows using a shadow volume and a stencil buffer, 2002.

[BWPP04] Jirí Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, September 2004.

[CDL+07] Simon Cox, Paul Daisey, Ron Lake, Clemens Portele, and Arliss Whiteside (editors). OpenGIS® Geography Markup Language (GML) implementation specification. `http://portal.opengeospatial.org/files/?artifact_id=4700`, 2007.

[Cla76] James H. Clark. Hierarchical geometric models for visible-surface algorithms. *SIGGRAPH Comput. Graph.*, 10(2):267–267, 1976.

[Dim07] Rouslan Dimitrov. Cascaded shadow maps. `http://developer.download.nvidia.com/SDK/10/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`, 2007.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. `http://doi.acm.org/10.1145/1111411.1111440`, 2006.

[FvDFH96a] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics (2nd ed. in C): principles and practice.* Addison-Wesley Systems Programming Series. Addison-Wesley Longman Publishing Co., Inc., 1996.

[FvDFH96b] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Illumination and shading. In *Computer graphics (2nd ed. in C): principles and practice*, pages 721–814. Addison-Wesley Longman Publishing Co., Inc., 1996.

[FvDFH96c] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. List-priority algorithms. In *Computer graphics (2nd ed. in C): principles and practice*, pages 672–680. Addison-Wesley Longman Publishing Co., Inc., 1996.

[FvDFH96d] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Spatial-partitioning representations. In *Computer graphics (2nd ed. in C): principles and practice*, pages 548–557. Addison-Wesley Longman Publishing Co., Inc., 1996.

[FvDFH96e] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. Visibe-surface ray tracing. In *Computer graphics (2nd ed. in C): principles and practice*, pages 701–714. Addison-Wesley Longman Publishing Co., Inc., 1996.

[GKC07] Gerhard Gröger, Thomas H. Kolbe, and Angela Czerwinski. Candidate OpenGIS® CityGML implementation specification. `http://portal.opengeospatial.org/files/?artifact_id=22120`, 2007.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, May 1987.

[Gus07] Vladislav Gusev. Extended Perspective Shadow Maps (XPSM). `http://www.xpsm.org`, 2007.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HMS06] W. Hunt, W. R. Mark, and G. Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *IEEE Symposium on Interactive Ray Tracing 2006)*, pages 81–88, 2006.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer*

graphics and interactive techniques, pages 269–278, New York, NY, USA, 1986. ACM.

[Kli01] Alex Klimovitski. SSE/SSE2 toolbox solutions for real-life SIMD problems. `http://www.gamasutra.com/features/gdcarchive/2001E/Alex_Klimovitski3.pdf`, 2001.

[Kol07a] Thomas H. Kolbe. CityGML - 3d geospatial and semantic modelling of urban structures. `http://www.citygml.org/fileadmin/citygml/docs/CityGML_ETS4_2007-03-21.pdf`, 2007.

[Kol07b] Thomas H. Kolbe. What is CityGML? `http://www.citygml.org/1533/`, 2007.

[Mam89] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9(4):43–55, 1989.

[MB89] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. In *Proceedings of Graphics Interface 89*, pages 152–163. Canadian Information Processing Society, June 1989.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166, 1990.

[MBW08] Oliver Mattausch, Jirí Bittner, and Michael Wimmer. CHC++: Coherent Hierarchical Culling revisited. *Computer Graphics Forum*, 27(2):221–230, April 2008.

[Mic08] Microsoft. *DirectX SDK (March 2008) C++*, 2008.

[PC02] European Parliament and European Council. Directive of environmental noise. `http://ec.europa.eu/environment/noise/directive.htm`, 2002.

[SBGS69] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, September 1969.

[Sch05] Daniel Scherzer. Shadow mapping of large environments. Master's thesis,

Institute of Computer Graphics and Algorithms, Vienna University of Technology, August 2005.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. `http://doi.acm.org/10.1145/566570.566616`, 2002.

[Shi05] Peter Shirley. *Fundamentals of Computer Graphics*. B&T, second edition, 2005.

[Sub90] K. R. Subramanian. *A Search Structure based on K-d Trees for Efficient Ray Tracing*. PhD thesis, The University of Texas at Austin, December 1990.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3):153–164, 2001.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing 2006)*, pages 61–69, 2006.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004.