DISSERTATION

GPU Enhanced Algorithms for Radiosity and Shadow Volume Rendering

ausgeführt zum Zwecke der
Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

eingereicht an der
Universität für angewandte Kunst Wien

1. Beurteiler:
o.Univ.Prof. Dr.techn. Georg Glaeser
Ordinariat für Geometrie
Universität für Angewandte Kunst Wien

2. Beurteiler:
Ao.Univ.Prof. Dr.techn. Eduard Gröller
Institut für Computergraphik und Algorithmen
Technische Universität Wien

von
Günter Wallner

Wien, im Januar 2009

Ich erkläre hiermit,

dass ich die Dissertation selbstständig verfasst, keine andere als die angegebenen Quellen und Hilfsmittel benutzt und mich auch sonst keiner unerlaubten Hilfen bedient habe,

dass diese Dissertation bisher weder im In- noch im Ausland (einer Beurteilerin / einem Beurteiler zur Beurteilung) in irgendeiner Form als Prüfungsarbeit vorgelegt wurde,

dass dieses Exemplar mit der beurteilten Arbeit übereinstimmt.

September 15, 2009

Abstract

Computer image generation has been driven by two major factors: realism and interactivity. Whereas the former led to various global illumination algorithms to solve the problem of light interreflection as accurately as possible the latter motivated algorithms for the fast generation of shadows for direct illumination environments. Computing a global illumination solution is an inherently difficult task, since solutions to integral equations – which express the transport of radiance in a recursive manner – have to be computed and therefore requires expensive computation times for sophisticated solutions.

Modern graphics hardware and their programmable shading units along with the fact that GPUs (Graphics Processing Units) are scaling well beyond Moore's Law, prompted researches to adopt, or at least accelerate, some popular graphics algorithms, like raytracing and radiosity, to the parallel architecture of GPUs.

The research – summarized in this thesis – was motivated by the above mentioned factors. One part of this thesis is devoted to the description of an extended GPU radiosity solver for triangular meshes, which is also capable of handling diffuse transmission as well as multiple ideal specular reflections and transmissions. Adaptive subdivision is used to increase the accuracy of the radiosity solution. For this purpose a new technique, which uses hardware occlusion queries to determine shadow boundaries in image space, is proposed. Furthermore the concept of light distribution textures to incorporate goniometric light sources into the radiosity process, is introduced.

The adaptability of these textures to real-time environments and for the simulation of *effect lights* is discussed – along with a shadow volume algorithm for the creation of shadow profiles – in the second major part which deals with GPU algorithms for shadows and light simulation in interactive environments.

Kurzfassung

Realismus und Interaktivität waren und sind die zwei treibenden Faktoren für die Entwicklung von Algorithmen zur Bilderzeugung. Ersterer führte zu unterschiedlichen globalen Beleuchtungsalgorithmen um das Phänomen der Lichtausbreitung so akurat wie möglich zu simulieren, während letzterer die Entwicklung von schnellen Algorithmen zur Erzeugung von Schatten in interaktiven Umgebungen veranlasste. Das Problem der globalen Beleuchtung erfordert die Lösung von Integralgleichungen – die den Lichttransport in rekursiver Form ausdrücken – und ist folgedessen von sich aus eine schwierige und zeitaufwändige Aufgabe.

Moderne Graphikhardware mit ihren programmierbaren Shadereinheiten und die Tatsache, dass die Zunahme ihrer Rechenleistung die Prognosen von Moores Law überschreitet, eröffnete Forschern die Möglichkeit, populäre Grafikalgorithmen wie Raytracing und Radiosity, an GPUs (Graphics Processing Units) zu adoptieren bzw. zu beschleunigen.

Die in dieser Arbeit zusammengefasste Forschungsarbeit wurde durch die oben genannten Faktoren motiviert. Ein Teil dieser Arbeit beschreibt einen erweiterten GPU Radiosity Algorithmus, welcher auch diffuse Transmission als auch mehrfache ideal spiegelnde Reflexion und Transmission beherrscht. Um die Genauigkeit der Lösung zu erhöhen, wird Adaptive Subdivision eingesetzt. Zu diesem Zweck wird eine neue Technik, die Schattengrenzen mit Hilfe von Occlusion Queries im Bildbereich ermittelt, vorgestellt. Desweiteren wird das Konzept von Light Distribution Textures, zur Simulation goniometrischer Lichtquellen, eingeführt.

Diese Texturen können auch in Echtzeitumgebungen und für die Simulation von Effektlichtern eingesetzt werden. Dies wird, zusammen mit einem Schattenvolumenalgorithmus für die Erzeugung von Schattenprofilen, im zweiten Teil dieser Arbeit – welcher sich mit der Simulation von Schatten und Lichtquellen in Echtzeit beschäftigt – diskutiert.

Preface

This thesis summarizes the work which was carried out at the University of Applied Arts Vienna, Department of Geometry between 2005 and 2008. It consists of five papers which were published either in international journals or conference proceedings. Separate chapters are devoted to each of these papers:

Chapter 1 Günter Wallner, *Geometry of Real Time Shadows*, Scientific and Professional Journal of the Croatian Society for Geometry and Graphics, 10, Zagreb, Croatia, 2006

Chapter 2 Günter Wallner, *GPU Radiosity for Triangular Meshes with Support of Normal Mapping and Arbitrary Light Distributions*, Journal of WSCG, 16, Plzen-Bory, Czech Republic, 2008

Chapter 3 Günter Wallner, *Geometry of Arbitrary Light Distributions*, ICGG 2008: Proceedings of the 13th International Conference on Geometry and Graphics, Dresden, Germany, 2008

Chapter 4 Günter Wallner, *An Extended GPU Radiosity Solver Including Diffuse and Specular Reflectance and Transmission*, Accepted for publication in The Visual Computer

Appendix A Günter Wallner, *Force Directed Embedding of Hierarchical Cluster Graphs*, ROGICS 2008: Proceedings of the International Conference on Relations, Orders and Graphs: Interaction with Computer Science, Mahdia, Tunisia, 2008

The first paper delineates the application of shadow volumes to the creation of so called shadow profiles, which describe the progression of shadows over a specific time period. The creation of these profiles is accelerated by harnessing the computational power of programmable graphics hardware and by employing a dual-space approach for silhouette finding.

The second work was motivated by the work of Coombe et al. who published a paper about *Radiosity on Graphics Hardware* in which they describe a radiosity solver which completely runs on a GPU. The text at issue describes, inter alia, an implementation for a GPU radiosity solver for arbitrary triangular meshes, introduces a

new method to identify elements on which adaptive subdivision should be performed to improve the accuracy of the solution by determining shadow boundaries in image space and introduces the concept of light distribution textures to incorporate goniometric light sources into the radiosity process.

Light distribution textures are used to simulate hemispherical and omnidirectional light sources for real time rendering in the third paper. Reconstruction of the photometric solid from the texture is improved by using bicubic interpolation with Catmull-Rom splines (instead of linear interpolation). Such textures can also be used to simulate various effect lights (e.g. disco lights).

The fourth article expands the GPU radiosity solver to include diffuse transmission as well as specular reflectance and transmission. The solver is capable to handle multiple specular reflections with correct mirror-object-mirror occlusions. Furthermore it describes the inclusion of ambient overshooting and delineates how a simple proxy object with alpha masks can substitute complex geometry and thereby speed up the computation times. A revised and abridged version of the article presented here has been accepted as a full paper for *Computer Graphics International 2009* and for publication in a special edition of *The Visual Computer*.

The last paper, which is not directly connected with the other four, and therefore presented in the appendix, describes a force directed layout adjustment algorithm for hierarchical cluster graphs. Hierarchical cluster graphs impose finer levels of granularity – which may be controlled by the user – by grouping nodes according to certain criteria. Clusters and hierarchies thereof can be dynamically closed and opened with the presented technique. In addition a simple clustering algorithm which proved useful for semantic nets is discussed.

Acknowledgments

First, I would like to thank my supervisor Georg Glaeser for the guidance he provided throughout the process of writing this thesis. I am also grateful to Eduard Gröller for being my second supervisor. I have furthermore to thank my colleagues, especially Franz Gruber for his help on various geometric issues.

I would like to express my sincere gratitude to my family and friends for their support and encouragement throughout the years.

Special thanks go to the reviewers of my manuscripts, who with their valuable feedback, helped to improve the quality of the final publications. Last but not least I want to thank the internet community who provided great help and advice on implementation related issues.

Contents

1. Geometry of Real Time Shadows

Shadows provide important visual cues about the spatial relationship among objects. Shadow volumes are one way to generate sophisticated shadows for use in real time environments. This paper focuses on the geometric aspects which are involved in the creation of the shadow volume. Speed up techniques like shaders and dual space approaches for silhouette determination are discussed. Finally the application of the described methods in a software for shadow profile calculation is addressed.

1.1. Introduction

Shadows are an important part in computer graphics because they can reveal information that otherwise would not be ascertainable. Foremost, they reveal the spatial relationship between objects in the scene. They also disclose new angles on an object that otherwise might not be visible and they can also indicate the presence of off-screen objects. These and other visual functions of shadows in computer graphics are described by Birn in [6].

Shadow volumes have been first proposed by Crow in 1977 [9]. With the advent of modern day computer graphic cards, shadow volumes are now possible in real time. Heidmann [13] adapted Crow's algorithm to hardware acceleration. His method is now known as the z-pass method (because the stencil buffer is incremented/decremented when a polygon passes the depth test). However, the z-pass method does not work correctly if the near clipping plane intersects the shadow volume. Carmack [7] solved the problem by using z-fail testing (the stencil buffer is incremented/decremented when a polygon fails the depth test). The z-fail method still yields incorrect results if the shadow volume is intersected by the far clipping plane. This problem can be circumvented by moving the far clipping plane to infinity, as proposed by Everitt and Kilgard [10].

Shadow maps (introduced by Williams [26]) are an image based alternative to shadow volumes (which operate on the object geometry). In the meantime several different shadow map algorithms have been developed. Both methods have their benefits and drawbacks. For a comparison of the pros and cons of both methods see e. g. [25].

"Classic" shadow volume algorithms create hard shadows. A shadow region is divided into two parts: the region which is fully in shadow (umbra) and the region which is partially in shadow (penumbra). Hard shadows only consist of the umbra area. Soft shadow volume algorithms have been among others studied by Assarsson and Akenine-Möller [1, 2].

1.2. Assumptions and Definitions

The shadow volume algorithm requires that the shadow casting object must be a 2-manifold polygon mesh and free of non-planar polygons. 2-manifold means that every edge of the mesh must be shared exactly by two polygons. It is also useful to restrict oneself to triangular meshes, because modern graphics hardware is optimized for triangle rendering.

Furthermore all triangles must have the same winding order. For the following discussion a counter clockwise winding order and outward pointing normals are assumed.

A silhouette edge is an edge adjacent to one front-facing and one back-facing polygon. A polygon is called front-facing in respect to the light if the dot-product of its

normal and the vector from the light position and a point on the polygon is positive. Respectively a polygon is called back-facing in respect to the light if the dot-product is negative.

A border edge is an edge which is only adjacent to one face (which implies that the mesh is open). It should be noted that we can handle open meshes if we treat border edges as part of the silhouette. The silhouette is the set of all silhouette edges (and border edges).

1.3. Overview

I will first give an overview of the z-pass algorithm and then point out the differences in respect to the z-fail algorithm. The basic concept is to use the stencil buffer as a masking mechanism to prevent pixels in shadow from being drawn during the render pass for a particular light source [16]. First of all the stencil buffer is initialized with zero and the z-buffer is initialized with the depth values of the visible objects during a first rendering pass. In this pass only light independent attributes are considered (e.g. ambient light). Then the shadow volume is rendered with writes to the color buffer and depth buffer disabled. This is usually done in two steps. First, the front faces of the shadow volume (in respect to the camera position) are rendered and the stencil buffer is incremented each time the fragment passes the depth test. Second, the back faces are rendered. This time decrementing the value in the stencil buffer when a fragment passes the depth test. As shown in Figure 1.1, this leaves non-zero values in the stencil buffer wherever the shadow volume intersects a visible object. Figure 1.1 in addition shows why this approach fails, if the shadow volume intersects the near clipping plane.

As noted by Batagelo and Costa [4] the front faces must be rendered before the back facing polygons to avoid shadow counting overflow. That is, because under OpenGL the result of the increment and decrement functions is clamped to lie between 0 and the maximum unsigned integer value ($2^n - 1$ if the stencil buffer holds n bits) [20]. However, rendering the shadow volume geometry twice is a suboptimal solution. The OpenGL extension EXT_stencil_two_side [23] allows separate stencil states for front faces and back faces to be specified simultaneously. Therefore front faces as well as back faces can be rendered at once. Though this time it is not guaranteed that the front facing polygons will be rendered before the back faces. Consequently the feasibility exists that the stencil value for a particular pixel is decremented before incremented. We can account for that possibility by using another OpenGL extension, namely EXT_stencil_wrap [22], which allows stencil values to wrap when they exceed the maximum and minimum stencil values.

Several authors [5, 4, 7] proposed methods that cap the shadow volume at the near plane. However, these are computationally expensive and they suffer from robustness problems.

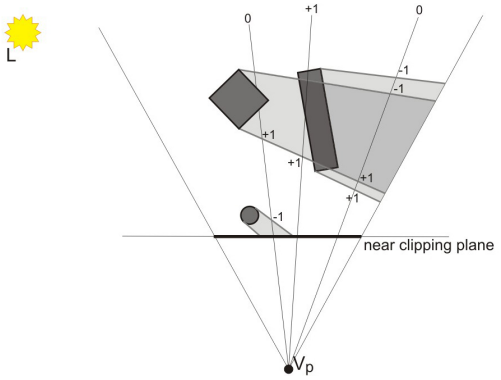


Figure 1.1.: The z-pass method. The values at the end of the rays represent the values left in the stencil buffer. Note that the stencil value of the leftmost ray is wrong due to the clipping of the shadow volume of the sphere at the near clipping plane.

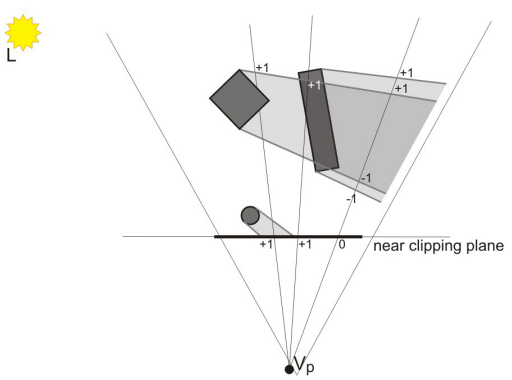


Figure 1.2.: The z-fail method. The values at the intersection of the ray and the near clipping plane represent the values left in the stencil buffer. This time the stencil value for the ray passing through the sphere is correct.

Carmack [7] and others therefore suggested the z-fail algorithm. Instead of counting the shadow faces in front of a particular pixel, the shadow faces behind are counted. This time the near clipping plane problem is avoided because shadow volume geometry between the eye and the pixel is nonrelevant. Figure 1.2 shows the z-fail approach. As already mentioned in the introduction the z-fail approach moves the near clipping plane problem to the far plane, which can be prohibited by using an infinite projection matrix (see Section 1.6).

1.4. Silhouette Detection

To calculate the shadow volume we first have to determine the silhouette of the shadow casting object. The so-called brute force method for detecting silhouette edges is to loop through all edges and check the dot-product of the adjacent triangles. Since silhouette detection is one of the two major bottlenecks (beside fill rate consumption), as pointed out by Kwoon [15], it is appropriate to use more sophisticated methods. Barequet et al. [3] developed a dual space approach for silhouette extraction in 3D and Hertzmann and Zorin [14] used a similar method but moved to four dimensions. Most recently Olson and Zhang [21] presented a paper about silhouette extraction in Hough space.

Because Hertzmann and Zorin [14] are concerned with non photorealastic rendering they determine the silhouette in respect to the viewpoint. However, in case of shadows the silhouette depends on the light position. Therefore the viewpoint must be substituted with the light position. Hertzmann and Zorin [14] build a dual surface by mapping each vertex $\mathbf{v}$ to a homogeneous point $\mathbf{v}' = (v_x, v_y, v_z, -(\mathbf{v} \cdot \mathbf{n}))$ where $\mathbf{n}$ is the unit normal vector of the vertex $\mathbf{v}$. The dual surface has the same con-

nectivity but different vertex positions. A dual edge e' of an edge $e = (\mathbf{v_1}, \mathbf{v_2})$ is a tuple $(\mathbf{v'_1}, \mathbf{v'_2})$. Let $\mathbf{L}$ be the homogeneous light position. An edge e belongs to the set of silhouette edges if $\mathbf{L} \cdot \mathbf{v'_1} >= 0$ and $\mathbf{L} \cdot \mathbf{v'_2} < 0$ or vice versa. Each $\mathbf{v'}$ is then normalized to make sure that each point of the dual surface lies inside the unit hypercube. This allows us to store each dual edge in a 4D variant of an octtree (I will call it hextree in the further discussion) as pointed out by Claes et al. [8]. At the highest level this hextree ranges from $(-1, -1, -1, -1)$ to $(1, 1, 1, 1)$. The space can be repeatedly divided into 16 smaller hextrees until a small enough partition is reached. A dual edge e' is then inserted into the smallest subcube which encloses $\mathbf{v'_1}$ as well as $\mathbf{v'_2}$.

Instead of using two bounding boxes per subcube to determine if the dual edges have to be verified [8] I use a different approach. For testing if an AABB[1] and a plane intersect in two dimensional space, the box diagonal which is most aligned with the normal of the plane has to be found first. Second the diagonals vertices ($\mathbf{v_{min}}$ and $\mathbf{v_{max}}$) are inserted into the plane equation. If the signs of result differ or at least one of them is zero, then the plane intersects the box [19]. Möller and Haines [19] also point out that the two vertices can be found directly. The signs of the components of the plane normal are used as a bit mask. If this mask is interpreted as a number it can be used as index to an array of AABB vertices. This approach can easily be extended to four dimensions. Each of the 16 vertices of a 4D cube is stored in an array so that the minimum vertex is located at index 0 and the maximum vertex at position 15. Instead of the plane normal we interpret the signs of the components of $\mathbf{L}$ as a bit mask. The index i of $\mathbf{v_{min}}$ can then be calculated as $i = 8 \cdot sgn(L_x) + 4 \cdot sgn(L_y) + 2 \cdot sgn(L_z) + sgn(L_h)$ where

$$sgn(x) = \begin{cases} 0 & x >= 0 \\ 1 & \text{otherwise.} \end{cases} \tag{1.1}$$

The $\mathbf{v_{max}}$ vertex can be found by inverting the bit mask. The dual edges of a subcube must only be tested if $\mathbf{L} \cdot \mathbf{v_{min}} >= 0$ and $\mathbf{L} \cdot \mathbf{v_{max}} < 0$ or vice versa.

Building the dual surface and inserting the dual edges into the hextree can be done once in a preprocessing step as long as the connectivity of the object does not change. Furthermore silhouette detection must only be performed if the object position changes in respect to the light position.

1.5. Shadow Volume Construction

Once the set of silhouette edges is determined the edges must be extruded to form the shadow volume. As described by Lengyel [16], no matter what finite distance

[1] AABB stands for Axis Aligned Bounding Box. Assuming an AABB is valid in our case because the hextree is axis aligned.

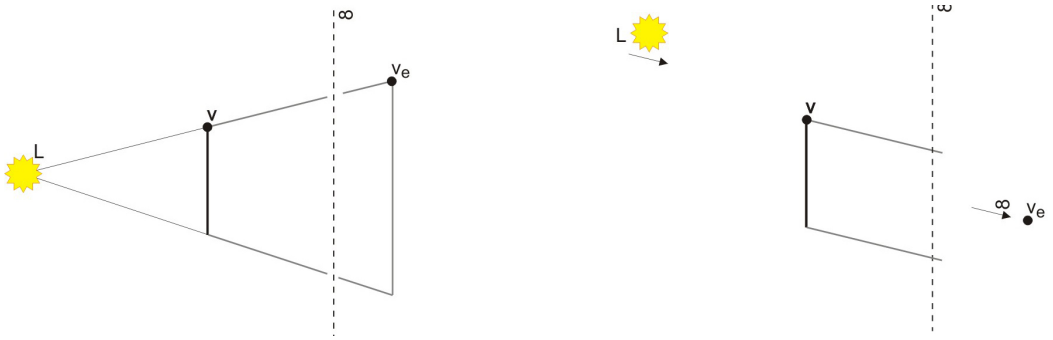


Figure 1.3.: Silhouette extrusion for a point light (left) and for a directional light (right)

silhouette edges are extruded, it is still possible that the shadow volume reaches not far enough to cast a shadow on every object in the scene that should intersect the volume. This problem worsens when the light source is very near to the shadow casting object, but it can be circumvented by using an infinite projection matrix. How this matrix can be obtained is described in Section 1.6.

In order for the z-fail algorithm to work correctly the shadow volume must be a closed volume where all polygons must have a consistent winding order. A complete shadow volume consists of: (1) the front cap (consisting of all front-facing polygons), (2) the extruded silhouette edges and (3) the back cap. It is notable that the extrusion of the geometry depends on the light source. For a point light the vertices of the silhouette edge must be extruded to infinity along the vector from the location of the point light to the vertex (see Figure 1.3). If $\mathbf{v} = (v_x, v_y, v_z, 1)$ is the position of the vertex to be extruded and $\mathbf{L}$ is the position of the point light then the extruded vertex $\mathbf{v}_e = (v_x - L_x, v_y - L_y, v_z - L_z, 0)$.

For a directional light all extruded points converge to a single point in infinity (see Figure 1.3) at position $(-L_x, -L_y, -L_z, 0)$. This implies that the back cap is not necessary for directional light sources. The back cap conventionally consisted of all back-facing polygons projected away from the light [10, 16]. But since the back cap is at infinity the shape does not matter [18]. The only constraint which remains is that the back cap must actually close the volume. This can be achieved with a simple triangle fan constructed from the extruded silhouette edges [18, 15].

The z-pass algorithm doesn't use caps, hence the incorrect results when the shadow volume intersects the near clip plane (see [10] for details) or the viewpoint is inside the volume. From this point it is clear that the z-fail method is computationally more expensive and should only be used when necessary. To determine whether the shadow volume is clipped by the near plane the near clip volume has to be constructed. The near clip volume is bounded by the planes which connect the near rectangle to the light position, as shown in Figure 1.4. The near rectangle is the area cut out of the near plane by the four side planes of the view frustum. Only an object which is inside this near clip volume can cast a shadow onto the near clipping plane.

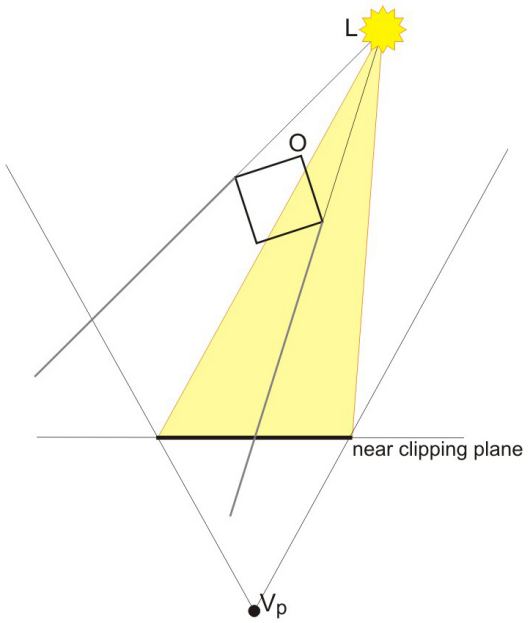


Figure 1.4.: An Object O is casting a shadow onto the near clip plane because it partially intersects the near clip volume (shaded)

For a comprehensive description see [16].

Silhouette edge extrusion can now be done on graphics hardware to remove the burden from the CPU. The following Cg vertex shader extrudes a vertex $\mathbf{v} = (v_x, v_y, v_z, v_w)$ if $v_w = 0$ otherwise the position is just passed through.

```
float4 lightToVertex = IN.position - lightPos;

float m = 1 - IN.position.w;
float4 outx = IN.position*(1-m) +
              lightToVertex*m;
outx.w = IN.position.w;

// transform position to homogenous clip space
OUT.HPOS = mul(ModelViewProj, outx);
```

Listing 1.1: Silhouette extrusion shader for a point light

`IN.position` is the vertex coordinate and `lightPos` is the position of the point light. If shaders are used, oneself has to take care to transform the vertex position into homogenous clip space, hence the multiplication with the modelview-projection matrix. For this approach to work correctly each vertex of the silhouette must be passed twice to the shader. Once with $v_w = 1$ and once with $v_w = 0$. The extrusion for a directional light looks similar.

1.6. Infinite Projection Matrix

The OpenGL projection matrix is defined as [20]:

$$P = \begin{vmatrix} \frac{2\cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2\cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2\cdot f\cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{vmatrix} \tag{1.2}$$

Where f is the distance from the viewer to the far clip plane, n the distance to the near clip plane and r and l are the respective distances to the left and right clip plane. t and b are the distances to the top and bottom clip plane. We can obtain the infinite projection matrix by calculating $P_\infty = \lim_{f\to\infty} P$ which yields

$$P_\infty = \begin{vmatrix} \frac{2\cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2\cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & -2\cdot n \\ 0 & 0 & -1 & 0 \end{vmatrix} \tag{1.3}$$

An infinite projection matrix reduces the depth buffer precision only marginally as pointed out by Everitt and Kilgard [10]. However, if you are concerned about this loss you can use Nvidia's NV_depth_clamp [24] extension. If depth clamping is enabled the near and far clipping plane are disabled for rasterizating geometry primitives.

1.7. Rendering

Here I present the necessary steps to render shadow volumes with OpenGL. First we render the scene with enabled depth writes, backface culling and with ambient lighting only (light independent attributes). This makes sure that the depth buffer is initialized with the correct depth values. Afterwards we disable writes to the depth buffer and turn off ambient lighting.

```
glEnable(GL_LIGHTING);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);

drawScene();

glDepthMask(GL_FALSE);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, zero);
```

The stencil mask has to be calculated separately for each light source.

```
for each light source
{
```

First we clean the stencil buffer, configure the stencil test so that it always passes and disable writes to the color buffer. We will make advantage of two side stencil testing so that we only have to render the shadow volume of each occluder once. Therefore the stencil operation is set to increment and decrement for front- and back-facing polygons respectively – when the depth test fails. Culling is also turned off because front as well as back faces must be rendered at the same time.

```
  glClear(GL_STENCIL_BUFFER_BIT);
  glEnable(GL_STENCIL_TEST);
  glStencilFunc(GL_ALWAYS, 0, ~0);
  glStencilMask(~0);

  glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

  glActiveStencilFaceEXT(GL_BACK);
  glStencilOp(GL_KEEP, GL_INCR_WRAP_EXT, GL_KEEP);
  glActiveStencilFaceEXT(GL_FRONT);
  glStencilOp(GL_KEEP, GL_DECR_WRAP_EXT, GL_KEEP);

  glDisable(GL_CULL_FACE);
  glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);
```

Now the shadow volume of each occluder in the scene is rendered. Afterwards culling is turned on and the stencil test is disabled. At that time the stencil buffer holds the correct information which pixels are in shadow and which aren't.

```
  for each occluder
  {
    renderShadowVolume(occluder);
  }

  glEnable(GL_CULL_FACE);
  glDisable(GL_STENCIL_TEST_TWO_SIDE_EXT);
```

The whole scene is now rendered again. This time the current light is enabled and configured (all light dependent attributes). Stencil testing is configured so that only pixels with a zero stencil value are rendered. Equal depth testing is used so that only visible fragments are updated. Since this pass adds to the ambient scene already in the color buffer additive blending must be enabled as well as writes to the color buffer. After rendering the scene blending is disabled and the depth function is restored to less depth testing.

```
  glEnable(light);
  configureLight(light);
```

```
  glEnable(GL_BLEND);
  glBlendFunc(GL_ONE, GL_ONE);
  glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

  glStencilFunc(GL_EQUAL, 0, ~0);
  glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
  glDepthFunc(GL_EQUAL);

  renderScene();

  glDisable(GL_BLEND);
  glDepthFunc(GL_LESS);
}
```

After the above steps have been carried out for all lights, stencil testing is disabled and writes to the depth buffer are enabled.

```
glDisable(GL_STENCIL_TEST);
glDepthMask(GL_TRUE);
```

1.8. Application: Shadow Profiles

I have succesfully applied shadow volumes in an application for calculating shadow profiles in real time. A shadow profile shows the casted shadow of an object over a specific time period. This is, for example, of concern for architects to find out how long the surrounding is obscured by a building. After providing the required information needed for computing the position of the sun[2] (latitude, date, time) and the time period the shadow profile is calculated.

The application can detect the silhouette either by brute force or with the above described dual space approach. If the graphics card supports vertex and fragment shaders, silhouette extrusion and per pixel lighting is performed on the GPU. Otherwise the CPU handles the extrusion and standard OpenGL lighting is used. Double sided stencil testing is performed if EXT_stencil_two_side is supported. The z-fail algorithm is only applied if necessary (see Section 1.3).

Figures 1.5 to 1.7 show some sample scenes. Table 1.1 shows the time needed for brute force silhouette detection for each scene and Table 1.2 for dual space silhouette detection, respectively. All measurements were taken on a Pentium 4 3.4Ghz processor with 1GB memory. For each scene a hextree with a fixed depth of four was chosen for the dual space approach.

[2] See [12] for a description of the calculation

Scene	Number of triangles[a]	∼time [ms]
Eiffel Tower	11353 (11155)	4.584
Industry Area	13615 (13585)	7.299
Uniqua Building	182038 (147296)	44.486
Uniqua Building	182038 (182038)	58.666

Table 1.1.: Performance with brute force silhouette detection

[a]First number: total triangles in the scene. Second number: triangles of shadow casting objects

Scene	Number of triangles	∼time [ms]
Eiffel Tower	11353 (11155)	4.236
Industry Area	13615 (13585)	5.799
Uniqua Building	182038 (147296)	39.331
Uniqua Building	182038 (182038)	48.872

Table 1.2.: Performance with dual space silhouette detection

1.9. Future Work

The results show that silhouette detection can greatly improve performance. As further work it would be interesting to see how hough space silhouette finding [21] can further speed up the process. At this time no techniques to reduce fill rate consumption were implemented. Lengyel [16] describes how OpenGLs scissor rectangle support can be used to cut down the fill rate penalty for rendering the shadow volumes. That is because the hardware does not generate fragments outside the scissor rectangle. The scissor rectangle can be applied on a per light basis or per geometry basis, as pointed out by Lengyel [17]. Everitt and Kilgard [11] suggest a depth bounds test for stencil writes. This idea is based on the observation that some depth values can never be in shadow, so incrementing and decrementing the stencil buffer is needless.

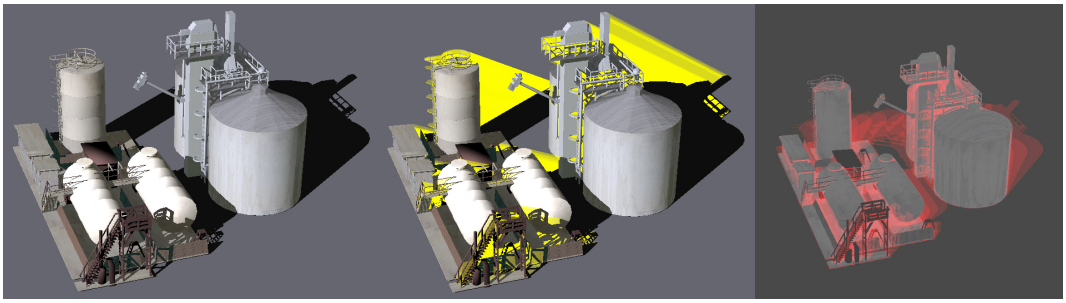


Figure 1.5.: left: Casted shadows of an industry area located at a latitude of 45.2° north on the 18th September at 3pm. middle: Visualization of the shadow volumes (yellow). right: The shadow profile of the scene over a time period of three hours (12pm until 3pm in 30 minutes time steps).

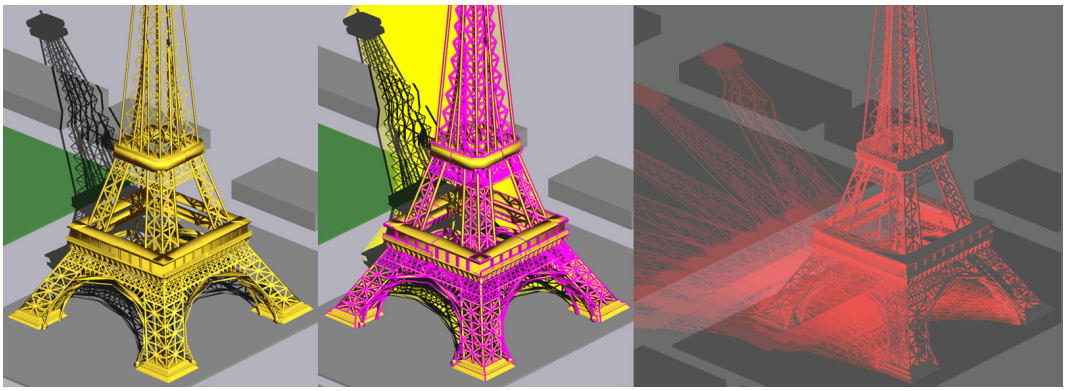


Figure 1.6.: left: Shadow of the Eiffel Tower in Paris (latitude of 48.8° north) on the 18th September at 2pm. middle: Visualization of the shadow volume (yellow) and the silhouette edges (pink). right: Shadow profile over a time period of four hours (10am until 2pm in 30 minutes intervals).

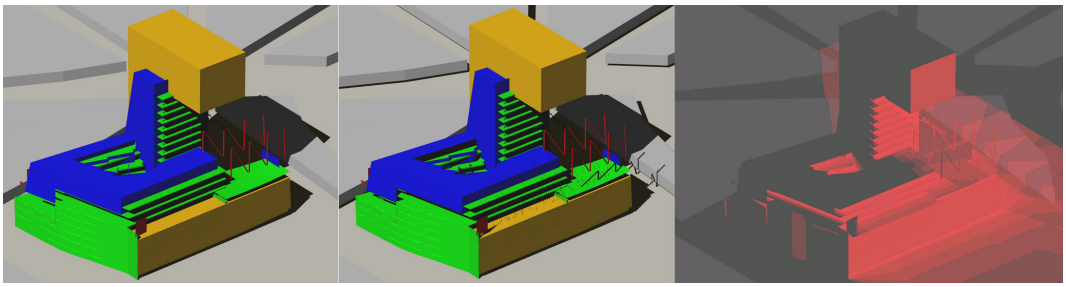


Figure 1.7.: Proposal for the Uniqua building in Vienna (48.2° north) by Hans Hollein. A color was assigned to each structural component. left: Only the facade (yellow) and the concrete (green) is casting a shadow. middle: The complete building is casting a shadow. right: The shadow profile over a time period of eight hours (9am until 5pm in 1 hour intervals) on the 18th September.

1.10. Conclusions

In this paper I have presented the necessary steps for a robust implementation of stencil shadow volumes. Stencil shadow volumes suffer mainly from two bottlenecks: (a) fill rate and (b) silhouette detection. The latter was discussed in Section 1.4. Modern graphics hardware can take over computations which formerly had to be performed on the CPU, e.g. silhouette extraction. Code snippets showed how stencil shadows can be implemented with OpenGL. Extensions to OpenGL provide further ways to improve performance.

1.11. Bibliography

[1] Akenine-Möller, T., Assarsson, U.: *Approximate soft shadows on arbitrary surfaces using penumbra wedges*. Proceedings of the 13th Eurographics workshop on Rendering pp. 297–306 (2002). Available online: `http://www.ce.chalmers.se/~uffe/softshadows_`

egrw.pdf

[2] Assarsson, U., Akenine-Möller, T.: *A geometry-based soft shadow volume algorithm using graphics hardware.* Siggraph Proceedings **22**, 511–520 (2003). Available online: http://www.cs.lth.se/home/Tomas_Akenine_Moller/pubs/soft_sig2003.pdf

[3] Barequet, G., Duncan, C.A., Goodrich, M.T., Kumar, S., Pop, M.: *Efficient perspective-accurate silhouette computation.* Proceedings of the 15th annual symposium on Computational geometry pp. 417–418 (1999)

[4] Batagelo, H.C., Junior, I.C.: *Real-time shadow generation using bsp trees and stencil buffers.* XII Brazilian Symposium on Computer Graphics and Image Processing pp. 93–102 (1999)

[5] Bestimt, J., Freitag, B.: *Real-time shadow casting using shadow volumes* (1999). Available online: http://www.gamasutra.com/features/19991115/bestimt_freitag_01.htm

[6] Birn, J.: *Digital Lighting & Rendering*, second edn. New Riders (2006)

[7] Carmack, J.: *E-mail to private list* (2000). Available online: http://developer.nvidia.com/attach/6832

[8] Claes, J., Fiore, F.D., Vansichem, G., Reet, F.V.: *Fast 3d cartoon rendering with improved quality by exploiting graphics hardware.* Proceedings of Image and Vision Computing New Zealand (IVCNZ) 2001 pp. 13–18 (2001). Available online: http://www.cs.utah.edu/npr/papers/Claes_IVCNZ2001.pdf

[9] Crow, F.: *Shadow algorithms for computer graphics.* SIGGRAPH Computer Graphics **11**(2), 242–248 (1977)

[10] Everitt, C., Kilgard, M.J.: *Practical and robust stenciled shadow volumes for hardware-accelerated rendering* (2002). Available online: http://developer.nvidia.com/object/robust_shadow_volumes.html

[11] Everitt, C., Kilgard, M.J.: *Optimized stencil shadow volumes.* Game Developer Conference Presentation (2003). Available online: http://developer.nvidia.com/object/GDC_2003_Presentations.html

[12] Glaeser, G.: *Der mathematische Werkzeugkasten*, second edn. Elsevier (2006)

[13] Heidmann, T.: *Real shadows, real time.* Silicon Graphics Inc. **18**, 23–31 (1991)

[14] Hertzmann, A., Zorin, D.: *Illustrating smooth surfaces.* Proceedings of the 27th annual conference on Computer graphics and interactive techniques pp. 517–526 (2000). Available online: http://mrl.nyu.edu/publications/illustrating-smooth/hertzmann-zorin.pdf

[15] Kwoon, H.Y.: *The theory of stencil shadow volumes* (2002). Available online: http:

//www.gamedev.net/reference/articles/article1873.asp

[16] Lengyel, E.: *The mechanics of robust stencil shadows* (2002). Available online: http://www.gamasutra.com/features/20021011/lengyel_01.htm

[17] Lengyel, E.: *Advanced stencil shadow and penumbral wedge rendering.* Game Developer Conference Presentation (2005). Available online: http://www.terathon.com/gdc_lengyel.ppt

[18] McGuire, M., Hughes, J.F., Egan, K.T., Kilgard, M.J., Everitt, C.: *Fast, practical and robust shadows* (2003). Available online: http://developer.nvidia.com/object/fast_shadow_volumes.html

[19] Möller, T., Haines, E.: *Real-Time Rendering*, first edn. A K Peters (1999)

[20] Neider, J., Davis, T., Woo, M.: *OpenGL Programming Guide*, first edn. Addison Wesley (1993)

[21] Olson, M., Zhang, H.: *Silhouette extraction in hough space.* Eurographics Proceedings **25** (2006). Available online: http://www.cs.sfu.ca/~haoz/pubs/06_eg_hough.pdf#search=%22silhouette%20extraction%20hough%20space%22

[22] OpenGL Extension Registry: *Ext_ stencil_ wrap* (2002). Available online: http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_wrap.txt

[23] OpenGL Extension Registry: *Ext_ stencil_ two_ side* (2003). Available online: http://oss.sgi.com/projects/ogl-sample/registry/EXT/stencil_two_side.txt

[24] OpenGL Extension Registry: *Nv_ depth_ clamp* (2003). Available online: http://oss.sgi.com/projects/ogl-sample/registry/NV/depth_clamp.txt

[25] Rege, A.: Shadow considerations. Available online: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Shadows.pdf

[26] Williams, L.: *Casting curved shadows on curved surfaces.* Siggraph Proceedings **12**, 270–274 (1978). Available online: http://accad.osu.edu/~waynec/history/PDFs/shadowmaps.pdf#search=%22casting%20curved%20surfaces%22

2. GPU Radiosity for Triangular Meshes with Support of Normal Mapping and Arbitrary Light Distributions

This paper describes an implementation of a progressive radiosity algorithm for triangular meshes which works completely on programmable graphics processors. Errors due to the rasterization of triangles are fixed in a post-processing step or with a fragment shader during runtime. Adaptive subdivision to increase the accuracy of the radiosity solution can be performed during render-time. Since we found that the gradient is not very robust to determine whether triangles should be subdivided or not, we propose a new technique which uses hardware occlusion queries to determine shadow boundaries in image space. The GPU implementation facilitates the simple integration of normal mapping into the radiosity process. Light distribution textures (LDTs) enable us to simulate a variety of real world light sources without much computational overhead. The derivation of such an LDT from a EULUMDAT file is described.

2.1. Introduction

Computer image generation has been driven by two major factors: realism and interactivity. The former has led to a variety of global illumination algorithms such as radiosity. Radiosity was first introduced to computer graphics by Goral et al. [16] to simulate the light interaction in strictly diffuse environments. The fraction of the radiant light energy leaving one particular surface which strikes a second surface is defined as the so-called form factor. These form factors can be obtained by computing the coifficients of a set of linear equations. Cohen and Greenberg [11] introduced the hemicube to support scenes with occluded surfaces, which where not considered in the original implementation. In [10], Cohen et al. presented a progressive refinement approach which eliminated the $O(n^2)$ storage requirements of former methods by calculating the form factors on-the-fly. Further speed ups can be gained by implementing the substructuring approach from Cohen et al. [9] where light is shot from a courser mesh to a finer sets of elements. Smits et al. [26] published a radiosity implementation which focuses on those parts of the scene which affect an image most. Although radiosity is usually restricted to diffuse surfaces, generalizations of the radiosity method which can handle general reflectance (e.g. by Sillion et al. [24]) and volumetric scattering due to participating media like smoke ([21]) have been proposed. A comprehensive treatment of the radiosity method can be found in [25] and [12]. However, for completeness the essential features are reviewed in Section 2.2.

Modern GPUs opened up a whole new research area, allowing researchers to compute a radiosity solution in much faster time or even at interactive rates. Keller [17] generates a particle approximation of the diffuse radiance in the scene using quasi-Monte Carlo integration. Afterwards, the graphics hardware renders an image with shadows for each particle which are considered as point light sources. Martin et al. [18] calculated a hierarchical radiosity solution on the CPU and refined the result by generating textures that represent the diffuse illumination. Nielsen and Christensen [19] accelerated the hemicube method using graphics hardware. Carr et al. [7] used floating point textures to store the result of the radiosity computation. Gautron et al. [15] adapted the irradiance cache ([30]) to graphics hardware. However, all of these publications used graphics hardware to accelerate certain elements of the radiosity solution. Coombe et al. [13] finally proposed a progressive radiosity implementation which worked solely on the GPU.

This paper follows the approach by Coombe et al. but extends it to arbitrary triangular meshes. Further contributions are the inclusion of normal mapping into the radiosity process, the support of arbitrary light distributions due to the use of light distribution textures (LDTs) and new way to determine shadow boundaries for adaptive subdivision. Figure 2.1 shows a radiosity solution obtained with our method. The reminder of this paper is structured as follows. Section 2.2 reviews in short the basics of radiosity and the progressive radiosity approach and Section 2.3 describes our implementation. Adaptive subdivision is explained in Section 2.4. We conclude the paper by presenting results and sample images (Section 2.5) as well as future

Figure 2.1.: GPU radiosity solution of a scene with 6534502 elements distributed over 13627 triangles.

work (Section 2.6).

2.2. Progressive Radiosity

The radiosity method evaluates the intensity (or radiosity) of discrete points and surface areas in a diffuse environment. The radiosity B_i of an element i is given by [16]

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \tag{2.1}$$

where E_i is the emission, ρ_i the reflectivity and F_{ij} the form factor between element i and j. F_{ij} is purely geometrical in nature and describes the fraction of energy leaving element j impinging on element i. If using the disc approximation of Wallace

et al. [29] to the differential form factor equation (Figure 2.2), F_{ij} is given by

$$F_{dA_i,A_j}(= F_{ij}) = A_j \sum_{i=1}^{m} \frac{\cos(\phi_i)\cos(\phi_j)}{d^2\pi + \frac{A_j}{m}} \tag{2.2}$$

where m is the number of sampling points on A_j. As noted by Coombe et al. [13] this disc approximation reduces artifacts between adjoining faces exhibited by the original form factor formulation [16] when used in conjunction with projection methods, like the hemicube approach by Cohen and Greenberg [11]. To assure conversation of energy in a closed environment the sum of all form factors for a given element i is equal to unity:

$$\sum_{j=1}^{n} F_{ij} = 1 \quad \text{for } i = 1 \ldots n \tag{2.3}$$

Contrary to the conventional radiosity algorithm, where all the form factors for the entire scene are precalculated, form factors are calculated on-the-fly in a progressive radiosity solver. Furthermore, shooting is always performed from the element radiating the most light energy, since those typically have the greatest impact on the illumination, leading to a solution which converges quickly in regard to accuracy. Additionally an ambient radiosity term

$$A = R \sum_{j=1}^{n} \Delta B_j F_{ij}^{'} \quad \text{for any } i \tag{2.4}$$

was introduced by Cohen et al. [10] to estimate reflected light in the earlier iterations, yielding a more adequate illumination during early stages. ΔB_j represents the unshot radiosity. $F_{ij}^{'}$ is a first approximation to the form factor and is given by

$$F_{ij}^{'} = \frac{A_j}{\sum_{k=1}^{n} A_k} \quad \forall i \tag{2.5}$$

and the interreflection factor R is defined as

$$R = \left(1 - \frac{\sum_{k=1}^{n} \rho_k A_k}{\sum_{k=1}^{n} A_k}\right)^{-1} \tag{2.6}$$

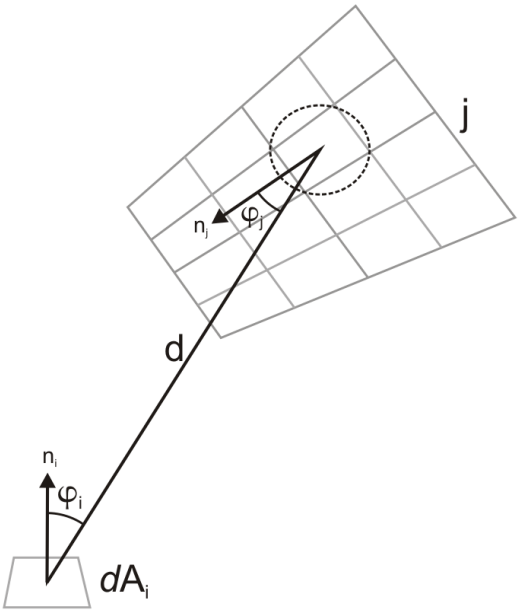


Figure 2.2.: The form factor between a differential area dA_i and a polygon j which is divided into m sections. Each section gets approximated by a disc.

2.3. Implementation

This section describes our radiosity implementation in detail. For each triangle two 32bit RGBA floating point textures are stored which hold the radiosity and residual energy respectively. The RGB components are used to store the illumination and the alpha channel is used to determine if a particular texel of the texture is occupied by the triangle ($A = 1$) or not ($A = 0$).

In a preprocessing step each triangle is rendered orthographically into a framebuffer of size $(2^n - 2) \times (2^n - 2)$. During rendering an occlusion query is issued to retrieve the number of texels occupied by the triangle. The area of a single element of a triangle can then be obtained by dividing its area with the result of the occlusion query. Note that due to partially covered pixels the area of an element is slightly underestimated. However, we found that no significant error is caused by this. The texture coordinates are retrieved by multiplying the vertex coordinates with the modelview-projection matrix used for rendering and shifting the values to the range $[0, 1]$. The result is then centered in a texture of size $2^n \times 2^n$ to allow for interpolation in the post-processing step. Furthermore all textures are placed in a texture atlas of size $2^m \times 2^m$ to reduce the number of texture switches during the radiosity process and the number of readbacks during the next shooter selection. All textures have power-of-two dimensions to allow for mipmapping.

After the preprocessing step the progressive radiosity solver starts until the result has converged or a maximum number of iterations has been reached. At the beginning

of each iteration the next shooter is determined. To find the triangle with the highest residual power the nth level of the mipmap pyramid is constructed from each residual texture atlas using a fragment program and a ping-pong rendering scheme. This results in a texture where each texel corresponds to the averaged residual intensity ($I = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B$) of a triangle. The reasons for a fragment program are twofold. First, graphics hardware may or may not support hardware mipmapping for floating point textures. Second, only texels which are occupied by the triangle may influence the average. Therefore our fragment shader only averages texels whose alpha channel equals one. The alpha channel of the new texel is set to one if one of the four original texels alpha value is one. The values are read back and multiplied by the area of the corresponding triangle to retrieve the residual power. The resulting values are compared and the triangle with the highest residual power is chosen as next shooter. During the next shooter selection the ambient radiosity term from Equation 2.4 is calculated. Since the average residual energy of a triangle is evaluated nevertheless and the overall interreflection factor can be precalculated, the computational expense is negligible.

Once a shooter has been selected, all the elements of the triangle shoot their energy in turn. The selected triangle is rendered orthographically into a framebuffer with two color attachments. A fragment shader outputs the interpolated normals and world positions of this triangle. As suggested by Coombe et. al [13], substructuring ([9]) can be supported by constructing a lower resolution mipmap of the residual texture. In our case, we also construct the mipmap from the normal and the world position map. The resulting residual mipmap gets sampled and each texel whose alpha channel equals one shoots its energy.

To determine the visibility from the current shooter, we follow the approach of Coombe et. al [13] and render the scene from the point of view of the shooter using a stereographic projection into a visibility texture. The position and orientation of the shooter are retrieved from the world position and normal map. However, instead of using color-encoded IDs of the polygons, we store the depth values as proposed by Barsi and Jakab [2]. Based on the front (fp) and back clipping distances (bp) the depth value is calculated in a vertex shader as shown in Algorithm 2.1.

```
pos = mul(modelView, position);
float z = (-2*pos.z-bp-fp)/(bp-fp); // [-1..1]
float zDepth = -z/bp                // [0..1]
```

Listing 2.1: Vertex shader code for calculation of the depth value

Since only vertices are affected by the stereographic projection, several errors are introduced, especially near the equator of the hemisphere. For example, convex quads may get concave after the projection, which leads to rasterization artifacts. Working with depth values instead of polygon IDs eliminates dot artifacts[1], since

[1] Due to the limited resolution of the visibility map and errors introduced by the projection, nearby elements of the scene may be mapped to the same texel.

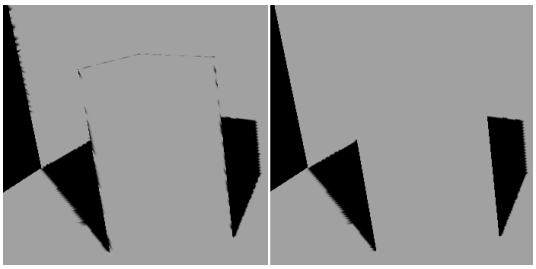


Figure 2.3.: Light Grey parts of the image are visible from the shooter, black ones aren't. If only one texel in the visibility texture is evaluated for depth correspondence artifacts appear near silhouette edges (left). Comparing also the neighboring texels removes those artifacts (right).

a tolerance value can be used when the visibility checks are performed later in the process. Triangles behind the hemisphere are culled away be checking against the plane defined by the position and normal of the shooter. For the remaining triangles, an occlusion query is issued.

Every triangle that might have received energy (triangles which pass the occlusion query test) is rendered orthographically to a framebuffer of size $(2^n - 2) \times (2^n - 2)$. However, instead of back-projecting the texels into the shooter's viewpoint, as done by Coombe et al. [13], the back projection is done in a vertex shader and the resulting position is passed to the fragment shader. This way the same error occurs during back projection as observed in the creation of the visibility texture. The fragment shader compares the depth value of the texel with the depth value stored in the visibility texture. We found that we can further reduce artifacts – mainly in areas of silhouette edges from the shooter's point of view – if we also check the neighboring texels in the visibility texture for correspondence with the – currently examined – texels depth (see Figure 2.3).

If the texel is declared visible, the form factor equation from Equation 2.2 is evaluated by the fragment program. The radiosity value is gained by multiplying the form factor, the shooters energy and the color as well as the reflectivity r of the receiver and adding it to the radiosity texture. Respectively the residual texture is updated by taking $1 - r$. After all texels of the shooter have shot their energy, the residual texture of the shooter is set to zero.

After the post-process (described in Section 2.3.1), the floating point textures are tone mapped using either a simple exposure function or a GPU implementation of the global tone mapping operator from Reinhard [20].

2.3.1. Rasterization of Triangles

According to the OpenGL specification [23] polygons and line segments are rasterized differently. For lines OpenGL uses a "diamond-exit" rule. This means that for each

fragment f with center at window coordinates x_w and y_w a diamond shaped region R_f is defined as

$$R_f = \{(x, y) | \|x - x_w\| + \|y - y_w\| < \frac{1}{2}\} \tag{2.7}$$

A good description of OpenGL's line rasterization can be found in [27]. For polygons OpenGL follows the point-sampling rule. Only fragments which centers lie inside the polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge (see [23] for details). However, we are not concerned about the exact details because those fragments get rasterized by line-rasterization anyway. Figure 2.4 shows the rasterization of a triangle. Since not all fragments – which are needed for texturing – are rasterized (these are shown red in Figure 2.4), the missing fragments are interpolated from the neighbor intensities in a post-processing step. To reduce artifacts due to rasterization, two steps are taken. First, every triangle is rendered twice. One time the polygon itself and next the outline with a line width of 1. It should be noted that using a line width greater than 1 leads to artifacts, since more than one fragment of the line has the same texture coordinate assigned, therefore pointing to the same location in the radiosity map. Second, after the radiosity solver has finished a textured quad is rendered orthographically to a framebuffer at the same resolution as the assigned radiosity texture to establish a one-to-one correspondence with the fragments of the framebuffer. A fragment program linearly interpolates the intensities for fragments which neighbor at least one fragment whose alpha channel is one. Only fragments occupied by the triangle are considered for interpolation. Since the textures are interpolated linearly for rendering, this is done twice, using a ping-pong technique. Fragments produced by this step are marked with black (first iteration) and blue (second iteration) dots in Figure 2.4. These fragments are only used for display purposes, therefore they are neither considered in the radiosity process nor do they alter the size of a triangle.

2.3.2. Light Distribution Textures

To include arbitrary light distributions into the radiosity process, we propose a so called light distribution texture (LDT). These textures can be derived from a EULUMDAT file or any other similar photometric file format. An English translation of the EULUMDAT specification can be found at [1]. Concordant do the specification we denote the number of C-planes as m_c. The number of light intensities in a C-Plane (vertical planes through the light distribution) is designated as n_g. Figure 2.5 shows the light distribution curve of a luminaire and it's 3D representation.

Although the EULUMDAT file stores photometric values and the radiosity method works with radiometric values, the normalized light distribution can be used as is. We can show that the radiant intensity $I_e = kI_v$, where k is some constant and I_v the luminous intensity (an in-depth treatment of lighting engineering can be found, for example, in [14]).

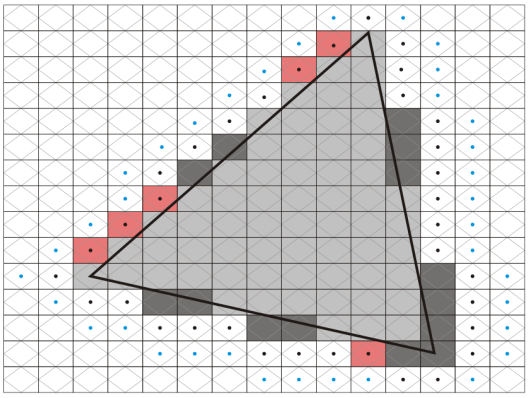


Figure 2.4.: Rasterization of a triangle with OpenGL. Light gray fragments are produced by polygon-rasterization. Dark gray rectangles depict fragments which were produced additionally by line-rasterization. Red fragments represent fragments which would be needed for GL_NEAREST texture sampling but have not been rasterized.

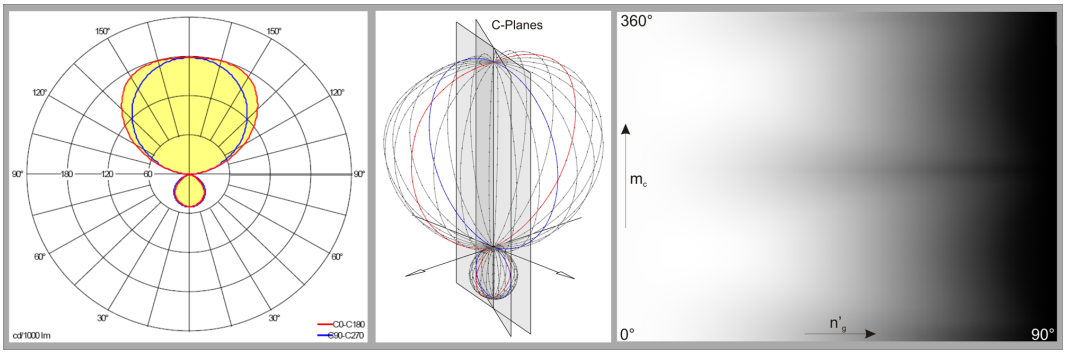


Figure 2.5.: The light distribution of a Zumtobel KAREA-S luminaire (left) and its 3D representation with $m_c = 24$ (middle). The red line depicts the intersection of the light distribution with the plane $C0/180$ and the blue line with plane $C90/270$ respectively. The texture derived from the luminaire's light distribution is shown on the right.

Proof. The radiant intensity can be written as

$$I_e = \frac{d\phi_e}{d\Omega} \tag{2.8}$$

where ϕ_e is the radiant flux and Ω the solid angle, and the luminous intensity can be written as

$$I_v = \frac{d\phi_v}{d\Omega} \tag{2.9}$$

Furthermore the luminous flux ϕ_v is defined as

$$\phi_v = K_m \int_{\lambda=380nm}^{780nm} \phi_{e_\lambda} V(\lambda) d\lambda \tag{2.10}$$

For a monochromatic lightsource we can reduce Equation 2.10 to

$$\phi_v = K_m V(\lambda) \phi_e \tag{2.11}$$

where $K_m = 683\ lmW^{-1}$ is the sensitivity of the eye at 555 nm and $V(\lambda) = 1$ for photopic vision (these values can be gained from the photopic vision curve $V(\lambda)$). For values of $V(\lambda)$ refer, e.g. to [14]. Substituting into Equation 2.8 we get

$$I_e = \frac{\frac{1}{683} d\phi_v}{d\Delta\Omega} = \frac{\frac{1}{683} I_v d\Delta\Omega}{d\Delta\Omega} = \frac{1}{683} I_v \tag{2.12}$$

□

An LDT stores the light distribution of a luminaire of one half space and has dimension $n'_g \times m_c$ where n'_g is the number of intensities of a C-Plane in one half-space. The intensity values are retrieved from the light distribution and divided by the maximum intensity value I_{max} to normalize the values to the range [0..1]. These values are written into the texture, where each horizontal line represents the intensity values of a C-Plane. The relationships are shown in Figure 2.5. Texture sampling is set to linear to automatically interpolate between the discrete measurements. To assure continuity at the boundary of 0° and 360°, the texture wrap mode in v-direction is set to GL_REPEAT. By storing only the normalized intensity distribution the texture can be reused for luminaires with the same light distribution but different intensities.

To access the LDT, the azimuth ϕ_r and elevation ϕ_n of the vector $\mathbf{d}$ with respect to the reference system of the light source given by $(\mathbf{n_0}, \mathbf{r_0}, \mathbf{u_0})$ is determined. We use the subscript 0 to denote unit vectors. Figure 2.6 shows a geometrical representation of the problem. Normalizing the angles to the range $[0, 1]$ yields

$$x_t = 1 + \frac{\min(\phi_n - \pi/2, 0)}{(\pi/2)} \tag{2.13}$$

$$y_t = \begin{cases} 1 - \frac{0.5\phi_r}{\pi} & u_0 \cdot \mathbf{d} \leq 0 \\ \frac{0.5\phi_r}{\pi} & \text{otherwise} \end{cases} \tag{2.14}$$

as texture coordinates (x_t, y_t). According to Sillion et al. [24] the energy d^2E emitted by a differential area dA_i around a point T_i in the direction of unit vector $\mathbf{d_0}$ and

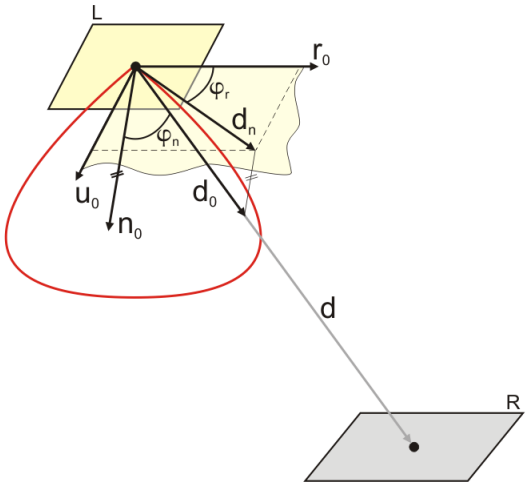


Figure 2.6.: The texture coordinates of the LDT for a given direction vector $\mathbf{d}$ between the lightsource L and receiver R depend on the azimuth ϕ_r and elevation ϕ_n of this vector. The light distribution is depicted as red curve.

falling on a differential area dA_j around a point T_j is then given by

$$d^2E = I(T_1, \mathbf{d_0}) \frac{cos(\phi_j)cos(\phi_i)}{\mathbf{d} \cdot \mathbf{d}} dA_j dA_i \tag{2.15}$$

where $I(T_1, \mathbf{d_0})$ is the intensity leaving the surface. In our case the intensity is retrieved by sampling the LDT at position (x_t, y_t) and multiplying it with I_{max}.

2.3.3. Normal Mapping

Inclusion of normal mapping [8] into the radiosity process is straightforward. Instead of taking the interpolated vertex normals for calculation of ϕ_j the perturbed normal is used. If the normals stored in the normal map are given in tangent space and since the light calculation is handled in world space, the vector $\mathbf{d}$ has to be transformed appropriately into tangent space. For static scenes the required tangent vectors can be calculated during the preprocessing step. Figure 2.7 and Figure 2.9 show results obtained with normal mapping.

However, it is worthy to note that if normal mapping and multitexturing is used simultaneously the resolution of the radiosity texture should correspond to the resolution of the normal map. Otherwise artifacts will appear because the result of light calculation does not overlay correctly with the texture of the object. Note also that illumination may change if normal mapping is used, because the shooting order need not necessarily be the same as without normal mapping.

Figure 2.7.: Radiosity solution of a simple box. Once without normal mapping (left) and one time with normal mapping enabled (right).

2.4. Adaptive Subdivision

The accuracy of the radiosity solution depends very much on the underlying mesh. As noted by several authors (e.g. [12]) uniform subdivision is not the best approach for radiosity, since some areas may be undersampled and others oversampled. Furthermore a too coarse mesh can introduce shadow leakage ([4, 6]). Several techniques to identify elements that require subdivision have been proposed (see [12] for an overview). For example, Vedel and Puech [28] subdivide if the gradient of the radiosity values varies more than a certain threshold. Campbell [5] splits an element perpendicularly to the line connecting the maximum and minimum points of an element, if the difference between the extrema exceeds a certain threshold. Campbell and Fussell [6] suggested a geometrical approach where the receiver polygon is tested against the shadow volume, generated by the light source and the occluding surfaces. However, the method is computationally expensive and does not scale well to complex scenes. We therefore propose the following method to determine if an element should be subdivided or not, by rendering the scene three times from the point of view of the shooter with a stereographic projection.

Step 1 Render the scene without depth testing and with occlusion queries enabled. This gives the complete number of rasterized fragments n_r for a triangle (independent from rendering order).

Step 2 Render the scene with depth testing enabled to initialize the depth buffer.

Step 3 Render the scene with depth testing and GL_LEQUAL as depth function and occlusion queries enabled. This yields the number of visible fragments n_v.

If $n_r \neq n_v$ there has to be a shadow boundary on this triangle. The triangle is subdivided if $b_l \leq n_v/n_r \leq b_u$ where $0 \leq b_l \leq 1$ and $0 \leq b_u \leq 1$ are the lower and upper threshold respectively. This avoids subdivision of triangles where the shadow

Figure 2.8.: The scene consists of 9012 triangles which are divided into 4259072 elements. The street lamp is simulated with a standard Lambertian light, the head and taillights are simulated with four spotlights. Each triangle was assigned a 32×32 radiosity texture and shooting was done from the third mipmap level.

Figure 2.9.: A scene illuminated by a Zumtobel wallwasher. There are 18436 triangles in the scene yielding 8440590 elements. The left image shows the scene without normal mapping, the middle and right image where rendered with normal mapping. The right image is a close-up view of the statue showing the reflecting light from the wall.

boundary is short.

In our implementation, we account for subdivision before shooting the first time from a triangle. If the area of the shooter is small, we found that a sufficient trade-off. In such a case, the rendering of the visibility texture can be combined with the steps outlined above. We follow the suggestion of Baum et al. [3] and use regular refinement for subdivision of triangles. Newly introduced triangles are tested again for subdivision until a maximum subdivision level has been reached or the subdivision criterion is not fulfilled. To avoid linear interpolation artifacts due to introduced T-vertices, these vertices are fixed with bisection refinement in regard to the balance criterion of Baum et al. [3]: the subdivision level of the neighboring elements should not differ more than one. If a triangle is subdivided, the radiosity and residual texture of the parent triangle is copied down to the child triangles using linear interpolation. Since subdivision is done before actually shooting, no reshooting as for example in [10] is performed.

Uniform	TS	t_{pp} [sec]	t_r [sec]	n_t	n_e	IT
box$_{nm}$	256(4)	0.96	28.37	42	1365280	16
box	32(3)	0.5	0.92	42	20120	16
bus	32(3)	8.84	58.8	8798	4192282	16
museum	32(3)	8.78	89.98	13627	6534502	10
statue	32(3)	13.45	104	16028	7683931	8

Adaptive	TS	t_{pp} [sec]	t_r [sec]	n_t	n_e	IT
bus	32(3)	8.84	72.26 (2.14)	9012	4259072	16
statue	32(3)	13.61	138.49 (14.43)	18288	8392967	8
statue$_{nm}$	32(3)	27.43	164.94 (33.0)	18436	8440590	8

Table 2.1.: Performance for uniform and adaptive meshing for different scenes

2.5. Results

The presented method was implemented with C++, OpenGL and the Cg shading language from NVidia. Table 2.1 shows information about the examples used throughout this paper. It lists the used radiosity texture size TS along with the mipmap level used for shooting (in brackets), the time consumed by the radiosity solver including the post-process and simple exposure tone mapping t_r and the time for setup and preprocessing (loading of scene geometry, calculation of tangent vectors, initializing of the texture atlas etc.) t_{pp}. Furthermore, the number of triangles n_t and the number of elements n_e as well as the number of iterations IT (an iteration includes shooting from all elements of a triangle) are listed. The subscript nm denotes that normal mapping has been used. The time in brackets represents the portion of t_r required for adaptive subdivision of the mesh. Except of the statue scene, all scenes have reached more than 88% convergence for the given number of iterations. All measurements where taken on a Intel Core2 CPU with 2.13 GHz with a Geforce 8800GTS with 640MB DDR3 Ram.

Performance analysis of the code showed that most time was consumed for rendering the receiver triangles. This is evident since this function is dependent on the result of the occlusion query to determine visibility. Additionally, setting the appropriate parameters of the orthographic projection for each triangle requires context switches of the fragment program. The performance of the current implementation is mainly limited by the available texture memory of the GPU. Once too many textures have to be maintained, texture memory thrashing can be noticed.

2.6. Conclusion and Future Work

We presented a GPU implementation of progressive radiosity for triangular meshes. The rasterization of triangles is the major problem to overcome. We solve this by

rendering the triangle itself and the outline of the triangle. The remaining artifacts are eliminated in a post-processing step or can be fixed during runtime with a fragment shader. Furthermore, we demonstrated the inclusion of normal mapping into the radiosity process, which yields more sophisticated results. Arbitrary light distributions can also be simulated with the help of light distribution textures.

The ample use of occlusion queries for determining visibility and shadow boundaries requires an elaborate algorithm to avoid stalling of the graphics pipeline. We are currently optimizing our implementation in this regard. Currently only one texture size is used for all triangles in the scene – independent from the actual size of a triangle. However, for scenes consisting of objects with rather coarse and fine meshes, this is suboptimal, since some triangles inevitable get undersampled or oversampled respectively.

We aim to include general reflectance distributions by means of BRDFs, as published by Sillion et al. [24], in our method. To allow for efficient reconstruction of the BRDF during runtime we are investigating the approach by NVidia [31]. To account for diffuse transmission the inclusion of a backward diffuse form factor [22], which denotes the fraction of energy leaving a surface from its back side and impinging on another surface, is considered.

2.7. Bibliography

[1] Ashdown, I.: English translation of the eulumdat specification. Available online: `http://www.helios32.com/Eulumdat.htm`

[2] Barsi, A., Jakab, G.: *Stream processing in global illumination.* Proceedings of 8th Central European Seminar on Computer Graphics (2004)

[3] Baum, D.R., Mann, S., Smith, K.P., Winget, J.M.: *Making radiosity usable: automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions.* In: SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques, pp. 51–60. ACM Press, New York, NY, USA (1991)

[4] Bullis, J.M.: Models and algorithms for computing realistic images containing diffuse reflections. Master's thesis, Dept. of Computer Science, Univ. of Minnesota (1989)

[5] Campbell, A.T.: Modeling global diffuse illumination for image synthesis. Ph.D. thesis, The University of Texas at Austin, Austin, TX, USA (1992)

[6] Campbell, A.T., Fussell, D.S.: *Adaptive mesh generation for global diffuse illumination.* In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pp. 155–164. ACM Press, New York, NY, USA (1990)

[7] Carr, N.A., Hall, J.D., Hart, J.C.: *Gpu algorithms for radiosity and subsurface scattering.* In: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS

conference on Graphics hardware, pp. 51–59. Eurographics Association, Aire-la-Ville, Switzerland (2003)

[8] Cohen, J., Olano, M., Manocha, D.: *Appearance-preserving simplification.* In: SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, pp. 115–122. ACM Press, New York, NY, USA (1998)

[9] Cohen, M., Greenberg, D., Immel, D., Brock, P.: *An efficient radiosity approach for realistic image synthesis.* Computer Graphics and Applications **6**(3), 26–35 (1986)

[10] Cohen, M.F., Chen, S.E., Wallace, J.R., Greenberg, D.P.: *A progressive refinement approach to fast radiosity image generation.* In: SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, pp. 75–84. ACM Press, New York, NY, USA (1988)

[11] Cohen, M.F., Greenberg, D.P.: *The hemi-cube: a radiosity solution for complex environments.* In: SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 31–40. ACM Press, New York, NY, USA (1985)

[12] Cohen, M.F., Wallace, J.R.: *Radiosity and Realistic Image Synthesis.* Morgan Kaufmann (1995)

[13] Coombe, G., Harris, M., Lastra, A.: *Radiosity on graphics hardware.* Technical report, Univ. of North Carolina, UNC TR03-020 (2003)

[14] Gall, D.: *Grundlagen der Lichttechnik - Kompendium.* Pflaum (2004)

[15] Gautron, P., Krivanek, J., Bouatouch, K., Pattanaik, S.: *Radiance cache splatting: A gpu-friendly global illumination algorithm.* Eurographics Symposium on Rendering (2005)

[16] Goral, C.M., Torrance, K.E., Greenberg, D.P., Battaile, B.: *Modeling the interaction of light between diffuse surfaces.* In: SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 213–222. ACM Press, New York, NY, USA (1984)

[17] Keller, A.: *Instant radiosity.* In: SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 49–56. ACM Press/Addison-Wesley, New York, NY, USA (1997)

[18] Martin, I., Pueyo, X., Tost, D.: *A two-pass hardware-based method for hierarchical radiosity.* Computer Graphics Forum **17**(3), 159–164 (1998). Available online: `citeseer.ifi.uzh.ch/martin98twopass.html`

[19] Nielsen, K.H., Christensen, N.J.: *Fast texture-based form factor calculations for radiosity using graphics hardware.* J. Graph. Tools **6**(4), 1–12 (2002)

[20] Reinhard, E.: *Parameter estimation for photographic tone reproduction.* J. Graph. Tools **7**(1), 45–52 (2002)

[21] Rushmeier, H.E., Torrance, K.E.: *The zonal method for calculating light intensities in the presence of a participating medium.* In: SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pp. 293–302. ACM Press, New York, NY, USA (1987)

[22] Rushmeier, H.E., Torrance, K.E.: *Extending the radiosity method to include specularly reflecting and translucent materials.* ACM Trans. Graph. **9**(1), 1–27 (1990)

[23] Segal, M., Akeley, K.: *The OpenGL Graphics System: A Specification (Version 2.0)* (2003)

[24] Sillion, F.X., Arvo, J.R., Westin, S.H., Greenberg, D.P.: *A global illumination solution for general reflectance distributions.* SIGGRAPH Comput. Graph. **25**(4), 187–196 (1991)

[25] Sillion, F.X., Puech, C.: *Radiosity and Global Illumination.* Morgan Kaufmann (1994)

[26] Smits, B.E., Arvo, J.R., Salesin, D.H.: *An importance-driven radiosity algorithm.* In: SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques, pp. 273–282. ACM Press, New York, NY, USA (1992)

[27] Sun, C., Agrawal, D., Abbadi, A.E.: *Hardware acceleration for spatial selections and joins.* In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 455–466. ACM Press, New York, NY, USA (2003)

[28] Vedel, C., Puech, C.: *A testbed for adaptive subdivision in progressive radiosity.* 2nd Eurographics Workshop on Rendering (1991)

[29] Wallace, J.R., Elmquist, K.A., Haines, E.A.: *A ray tracing algorithm for progressive radiosity.* In: SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques, pp. 315–324. ACM, New York, NY, USA (1989)

[30] Ward, G.J., Rubinstein, F.M., Clear, R.D.: *A ray tracing solution for diffuse interreflection.* In: SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, pp. 85–92. ACM Press, New York, NY, USA (1988)

[31] Wynn, C.: *Nvidia corporation. real-time brdf-based lighting using cube-maps* (2000)

3. Geometry of Arbitrary Light Distributions

Sophisticated simulation of lighting is crucial for plausible computer image generation. In many real time applications the simulation of light is restricted to directional, point and spot light sources. This paper presents a method to simulate hemispherical and omnidirectional light sources with arbitrary light distributions by means of light distribution textures (LDTs). These textures can be gained from photometric data files provided by manufacturers of luminaires. The derivation of LDTs from such files is presented. Reconstruction of the photometric solid from the LDT with bilinear and bicubic filtering is also discussed.

3.1. Introduction

Realistic simulation of lighting is necessary for plausible computer image generation. Many physical plausible rendering systems, like Lightscape, Lightworks, or Radiance, nowadays allow the inclusion of goniometric light sources[1] which permits the accurate simulation of real world luminaires. As pointed out by Albin and Peroche [1] such simulations are needed in architectural design of galleries, offices (certain lighting conditions have to be met by law), tunnels and so on. Furthermore the lighting industry relies on the accurate simulation of illumination caused by a luminaire before production of the luminaire actually starts.

However, in real time rendering direct illumination is still usually performed with point-, directional- and spot lights and, as it seems, not much work has been carried out in this area. In this paper we show the application of light distribution textures – a concept we introduced in [15] in regard to radiosity – to real time rendering.

The remainder of this paper is structured as follows. Section 3.2 introduces the necessary concepts and definitions. Related work in the field of computer graphics is reviewed in Section 3.3. Light distribution textures are covered in Section 3.4. Implementation details are provided in Section 3.5 and Section 3.6 presents results achieved with our method. Finally the paper is concluded in Section 3.7.

3.2. Concepts and Definitions

In *far-field* photometry a light source is regarded as a point source for which the inverse square law applies. The inverse square law states that the intensity of light radiating from a point source is inversely proportional to the square of the distance from the source. According to Ashdown [5] this assumption holds true for most architectural luminaires if the distance from the luminaire to the measurement point is at least five times the maximum width of the luminaire (five-times rule). Its photometric distribution is usually expressed as a *goniometric diagram* (see Figure 3.1). These diagrams represent a planar slice through the light distribution and therefore plot the relative intensity as a function of vertical angle specified in candela. A typical goniometric diagram depicts two perpendicular slices through the intensity distribution in polar coordinates, as noted by Cohen and Wallace [8]. Languénou and Tellier [11] suggested a method for interpolating smoothly between those two slices. However, for luminaires which exhibit a more complex light intensity distribution (LID) further slices are necessary.

The 3-dimensional extension of a goniometric diagram is referred to as *photometric solid* (see [4]). Such a photometric solid depicts the variation over vertical and horizontal angles simultaneously (refer to Figure 3.2).

[1] A goniometric light source is one which can emit widely varying amounts of light energy in different directions.

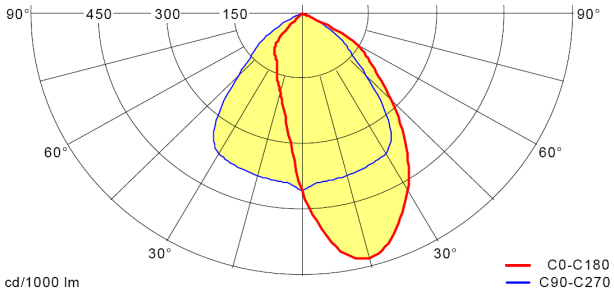


Figure 3.1.: Goniometric Diagram of an asymmetrical spotlight. The red and blue slice is also depicted in the photometric solid in Figure 3.2.

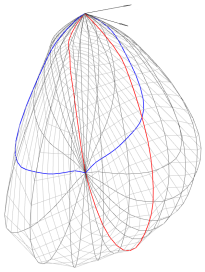


Figure 3.2.: Photometric solid of the asymmetrical spotlight.

The measurements of a source's LID are provided by the manufacturer in terms of a photometric file. Various photometric file formats exist, where – according to Ashdown [5] – EULUMDAT is the de facto industry standard in Europe, while IESNA LM-63 is used by North American lighting manufacturers. We will focus on EULUMDAT files in this paper, though LDT can also be gained from other photometric files. Unfortunately it seems that no official specification on the EULUMDAT standard is available on the web. However, an English translation can be found at [2]. In the remainder of this paper we will use the nomenclature of the EULUMDAT specification where m_c is the number of C-Planes (vertical slices) and n_g is the number of measurements per C-Plane.

3.3. Related Work

Goniometric diagrams where first introduced to computer graphics by Verbeck and Greenberg [14]. As already pointed out briefly in Section 3.2, Languénou and Tellier [11] showed how missing values can be interpolated from two perpendicular slices of the goniometric diagram, by projecting the direction vector onto the two planes and then performing an elliptic interpolation between the two retrieved intensity values. Zotti et al. [16] presented a method to approximate a luminaire with a combination of at most two OpenGL lights. Their method did not exploit programmable hardware, only OpenGL's built in light sources and therefore had shortcomings with lights that do not have spotlight characteristics. For global illumination renderer more work on this topic is available. For example, Albin and Peroche [1] proposed a method to reconstruct the light distribution from a goniometric diagram. To accomplish this, a locally supported kernel is associated with every measurement to avoid some problems with bilinear interpolation. They also cite more work in regard to global illumination.

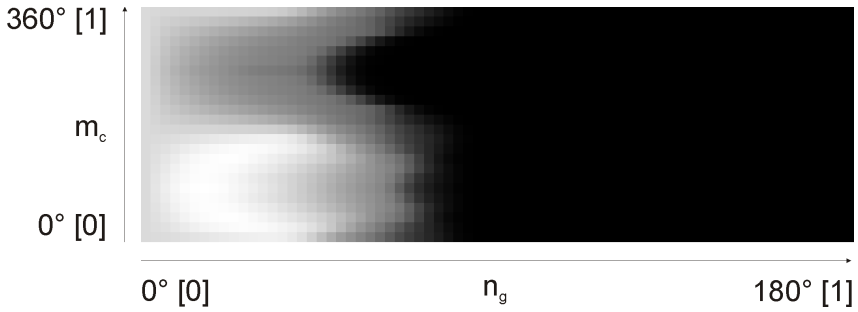


Figure 3.3.: Light distribution texture of the luminaire from Figure 3.2. Texture coordinates are given in parenthesis.

3.4. Light Distribution Textures

In this section we will first explain how an LDT can be derived from a light intensity distribution and then describe the necessary calculations to access such a texture.

An LDT stores the light distribution of a luminaire and has dimension $n_g \times m_c$. The intensity values are read from the photometric file and normalized to the range [0..1] by dividing through the maximum intensity I_{max}. Storing normalized values allows to control the intensity of a light source independently (e.g. for dimming). Figure 3.3 shows the LDT from the asymmetrical spotlight from Figure 3.2. In case of a rotationally symmetric light distribution only a one-dimensional texture has to be used. If the light distribution covers just one hemisphere, only the values from that hemisphere have to be stored.

To access the LDT to retrieve the light intensity for a given direction $\mathbf{d}$, the appropriate texture coordinates have to be calculated. For a rotationally symmetric LID and a light source with reference system $(\mathbf{n}_0, \mathbf{r}_0, \mathbf{u}_0)$[2] the angle θ between $\mathbf{d}$ and $\mathbf{n}_0$ has to be calculated and mapped to the range [0..1]. This is done by dividing θ by π for a omnidirectional light source or by $\pi/2$ for a hemispherical source. Consequently the texture coordinates are given by

$$\begin{aligned} s &= \frac{\arccos(\mathbf{n}_0 \cdot \mathbf{d})}{\pi} \\ t &= 0.5 \end{aligned} \tag{3.1}$$

For a luminaire with an asymmetric luminous flux distribution the issue is a little bit catchier, because the luminaire has to be orientated correctly in the scene. This is problematic, because – as noted by Ashdown [5] – various IESNA LM-series documents provide contradictory specifications on how the photometric solid is to be orientated with respect to the physical outline of a luminaire. See [5] for a more in-depth discussion on this topic. To obtain the correct C-plane for a non-rotationally symmetric LID, $\mathbf{d}_0$ is projected orthogonally onto the plane $(L, \mathbf{u}_0, \mathbf{r}_0)$. Afterwards the angle between the projected vector and $\mathbf{r}_0$ is calculated. Finally, we have to eval-

[2]The subscript 0 is used to denote unit vectors.

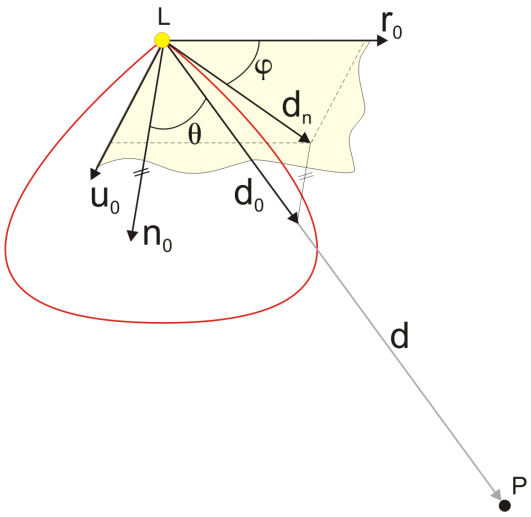


Figure 3.4.: Geometrical relationships which are necessary to sample the LDT for a given light source L and a point P. The light distribution is shown as red curve.

uate if $\mathbf{d}$ is in the positive or negative halfspace in respect to the plane $(L, \mathbf{n}_0, \mathbf{r}_0)$. The t-coordinate is therefore given by

$$t = \begin{cases} 1 - \frac{0.5\varphi}{\pi} & u_0 \cdot \mathbf{d} \leq 0 \\ \frac{0.5\varphi}{\pi} & \text{otherwise} \end{cases} \tag{3.2}$$

where

$$\varphi = \arccos(\mathbf{r}_0 \cdot ((\mathbf{d} \cdot \mathbf{r}_0)\mathbf{r}_0 + (\mathbf{d} \cdot \mathbf{u}_0)\mathbf{u}_0)) \tag{3.3}$$

The geometric situation is illustrated in Figure 3.4. By setting the interpolation mode of the texture to linear, missing values are automatically interpolated by graphics hardware. The texture wrap mode in s-direction is set to GL_REPEAT to ensure continuity between 0° and 360°.

3.4.1. Filtering

Mapping the light distribution to a two-dimensional texture has the advantage that two-dimensional filtering methods can be used to interpolate missing values.

Ashdown [3] points out that simple bilinear interpolation between the nearest measurement angles is probably adequate for all practical applications. However, in cases where only a small number of measurements is provided for a luminaire, interpolation artifacts may appear, since it gives only a piecewise linear approximation. Further problems with bilinear interpolation arise if the measurements are not taken regularly, as noted by Albin and Péroche [1]. In such cases higher-order interpolation schemes can be used to improve the sampling of the LDT. Bicubic filtering yields itself well to the task because (a) the interpolated result is smooth in all directions and (b) it can be performed very fast on programmable graphics hardware.

For instance, Bjorke [6] presented a method to perform bicubic filtering with the Mitchell-Netravali kernel in the fragment shader. Sigg and Hadwiger [13] presented a method to perform cubic b-spline filtering with linear texture fetches instead of repeated nearest-neighbor sampling (as done by Bjorke). This reduces the number of required texture samples and therefore increases the performance even further. Note that in case of LDTs an interpolating filter is required, since the provided data is certain to be part of the photometric solid[3]. Results for reconstructing the photometric solid with Catmull-Rom splines are provided in Section 3.6.

3.5. Implementation

The presented method can be easily included into existing surface reflection shaders (like e.g. Blinn-Phong reflection model) by replacing the constant light intensity with the intensity sampled from the LDT. The light intensity used for shading a surface point is therefore given by

$$B = I \cdot LDT(\theta, \varphi) \cdot \frac{1}{d^2} \tag{3.4}$$

where I is the absolute intensity of the light source, $LDT(\theta, \varphi)$ is the normalized intensity sampled from the LDT and $1/d^2$ accounts for the inverse square law of light (d being the distance from the lightsource). The sourcecode for accessing the LDT in a fragment shader is given in Listing 1.

We implemented the dual paraboloid shadow mapping method of Brabec et al. (refer to [7]) to show results in conjunction with shadows. It should be noted that any other shadowing technique which supports omnidirectional or hemispherical light sources, like Gerasimovs omnidirectional shadow mapping [10] or Shadow Volumes [9], can be used too.

A floating point off-screen buffer is used to allow high dynamic range lighting. On this buffer a tone mapping operation is performed to map the values to the displayable output range [0..1]. For the images in this paper we used a simple exposure function defined by $y = 1 - e^x$.

```
float3 d = normalize(lightPosition - pos);

// calculate s-coordinate
float cosi = -dot(lightNormal, d);
float phii = -min(acos(cosi) - PI, 0) / PI;

// calculate t-coordinate
float3 up = cross(lightNormal, lightRight);
float3 dn = normalize(dot(-d,lightRight)*lightRight +
   dot(-d,up)*up);
```

[3] This is not the case for the cubic b-spline filtering method presented in [13]. However, the method can be adopted to e.g. Catmull-Rom splines.

Figure 3.5.: Roman god mercury lit by a rotationally symmetric high-bay luminaire (left). The bilinear sampled light intensity from the LDT which is used for shading a surface point (middle) compared with the result of bicubic interpolation (right). White means maximum intensity whereas black is zero intensity. The goniometric diagram of the luminaire is depicted in Figure 3.6.

```
float cosr = dot(lightRight, dn);
float W = 0.5*(acos(cosr)/PI);

float cosu = -dot(up, d);
float f = max(0, sign(cosu));
float texV = (1-W)*f + W*(1-f);

// sample light intensity from the LDT
lightIntensity = tex2D(LDT, float2(1-phii,texV));
```

Listing 3.1: Cg fragment shader code for accessing a non-rotationally symmetric omnidirectional light distribution texture

3.6. Results

Figure 3.5 shows a scene which is lit by a circular high-bay luminaire which exhibits a *winged* emission pattern. Disco and effect lights can also be simulated with LDTs (see Figure 3.8). It is worth mentioning that similar effects can be reached with projective texture mapping (see [12]). However, in case of projective texture mapping the texture is only projected onto the surface, it does not influence the lighting calculation. Furthermore it is not possible to project the texture onto the whole halfspace at once. Figure 3.7 shows a room lit by four light sources. For the luminaire at the ceiling a compact downlight with extremely wide light distribution was used. For the lamp on the wall rotationally symmetric spotlights with a slightly anomalous distribution were used, and an asymmetrical light distribution of a downlight was assigned to the desk lamp. Average rendering times over 300 frames for different scenes on an Intel Core2 with 2.13 GHz, 3.5 GB RAM and a Geforce 8800GTS with 640MB RAM are given in Table 3.1. The table states the number of triangles n_t in the scene as well as the number of light sources n_l. The average rendering time of a frame if standard point light sources are used is denoted as t_s and the average rendering time with asymmetrical light distributions by means of bilinear interpolated LDTs is designated as t_g. The increase in rendering time is given in the

n_t	n_l	$t_s[ms]$	$t_g[ms]$	$inc[\%]$
96	1	2.106	2.161	2.61
	4	4.446	4.692	5.16
	7	7.300	7.465	2.26
1760	1	1.603	1.686	5.18
	4	3.423	3.729	8.94
	7	4.781	5.374	12.4
14764	1	2.292	2.473	7.9
	4	5.031	5.619	11.69
	7	7.906	8.863	12.1
114445	1	4.435	4.588	3.45
	4	10.051	10.322	2.7
	7	16.601	17.261	3.98

Table 3.1.: Average rendering times

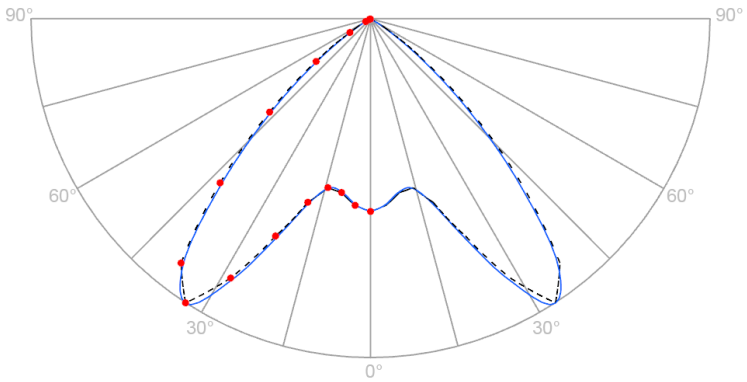


Figure 3.6.: Reconstruction of the photometric solid from the rotationally symmetric light distribution of the luminaire used in Figure 3.5.

column *inc*. In both cases the same render settings[4] were used, only the shader for calculating the lighting was interchanged. As shown, the increase in rendering time is tolerable and can be further reduced if only symmetrical light distributions are used.

To compare the quality of the reconstruction of the photometric solid from an LDT, we implemented Bjorkes [6] bicubic filtering method, because it allowed us to easily try different cubic splines by changing the filter weights in the look-up texture. To compare the results of various interpolation methods, the LDT was rendered to a five-times larger (floating point) texture. Afterward the contents of the texture were obtained and used for geometric reconstruction of the photometric solid. Currently the best results were achieved with Catmull-Rom splines. Figure 3.6 compares the results for a rotationally symmetric light distribution. Red points depict data present in the photometric data, which were omitted in the right half. The blue line shows bicubic interpolation with Catmull-Rom splines and the black dashed line linear

[4]This includes accumulating the contribution of each light source in a frame buffer object and subsequent tone mapping.

Figure 3.7.: A room lit by four goniometric light sources. Note the light distribution at the back wall, which does show a slightly anomalous spot light characteristic.

interpolation. Bilinear interpolation works well as long as the intensity does not change much over angular distance. Note, e.g. the bright white stripe in Figure 3.5 right, which corresponds to the intensity maximum at about 35° in the goniometric diagram. Such artifacts are not visible with bicubic interpolation since the derivatives are continuous over the photometric solid.

3.7. Conclusions

In this paper we showed the application of light distribution textures to real time rendering. LDTs allow the fast rendering of hemispherical as well as omnidirectional light sources with directionally dependent light distribution. They can easily be incorporated into existing surface reflection shaders. It should be stressed that in this paper we did not focus on physical plausibility. If this is desired, factors like e.g. the dependency of the luminous flux from the ambient temperature have to be considered. In addition LDTs allow the simulation of various effect lights. Time measurements have shown that the increase in rendering time is acceptable. However, there are still some code optimizations possible, which will result in further performance gains. Finally, different interpolation methods for reconstruction of the photometric solid

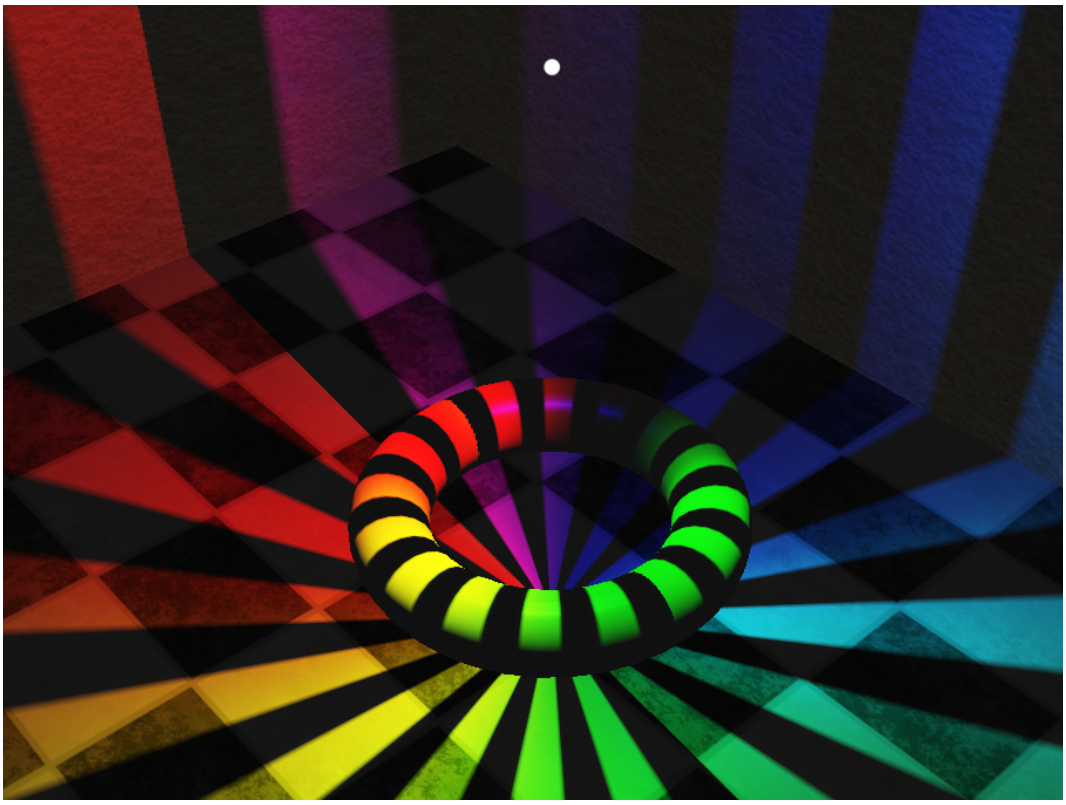


Figure 3.8.: LDTs can also be used to simulate various light effects, like disco lights. The white dot represents the position of the light source.

were discussed. As possible future work the viability of Albin and Péroche's method [1] to real time rendering should be investigated. We are currently experimenting with different kind of bicubic reconstruction filters. To compare the results we are using the Hausdorff Distance Metric, as suggested by [4]. It may be also from interest to compare those results with the ones achieved if using the method of Albin and Péroche.

3.8. Bibliography

[1] Albin, S., Peroche, B.: *Directionally dependent light sources.* In: Journal of WSCG, vol. 11. University of West Bohemia (2003)

[2] Ashdown, I.: English translation of the eulumdat specification. Available online: `http://www.helios32.com/Eulumdat.htm`

[3] Ashdown, I.: *Parsing the IESNA LM-63 photometric data file* (1998). URL `http://lumen.iee.put.poznan.pl/kw/iesna.txt`

[4] Ashdown, I.: *Comparing photometric distributions.* Tech. rep., Department of Com-

puter Science, University of British Columbia (1999)

[5] Ashdown, I.: *Thinking photometrically part II.* LIGHTFAIR 2001 Pre-Conference Workshop (2001)

[6] Bjorke, K.: *High-quality filtering.* In: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley Professional (2004)

[7] Brabec, S., Annen, T., Seidel, H.P.: *Shadow mapping for hemispherical and omnidirectional light sources.* In: Advances in Modelling, Animation and Rendering (Proceedings of Computer Graphics International), pp. 397–408 (2002)

[8] Cohen, M.F., Wallace, J.R.: *Radiosity and Realistic Image Synthesis.* Morgan Kaufmann (1995)

[9] Everitt, C., Kilgard, M.J.: *Practical and robust stenciled shadow volumes for hardware-accelerated rendering* (2002). Available online: `http://developer.nvidia.com/object/robust_shadow_volumes.html`

[10] Gerasimov, P.S.: *Omnidirectional shadow mapping.* In: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley Professional (2004)

[11] Languénou, E., Tellier, P.: *Including physical light sources and daylight in global illumination.* Proceedings of the Third Eurographics Workshop on Rendering pp. 217–225 (1992)

[12] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., Haeberli, P.: *Fast shadows and lighting effects using texture mapping.* SIGGRAPH Comput. Graph. **26**(2), 249–252 (1992). DOI http://doi.acm.org/10.1145/142920.134071

[13] Sigg, C., Hadwiger, M.: *Fast third-order texture filtering.* In: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, pp. 313–329. Addison-Wesley Professional (2005)

[14] Verbeck, C., Greenberg, D.: *A comprehensive light-source description for computer graphics.* IEEE Comput. Graph. Appl. **4**(7), 66–75 (1984). DOI http://dx.doi.org/10.1109/MCG.1984.275906

[15] Wallner, G.: *GPU radiosity for triangular meshes with support of normal mapping and arbitrary light distributions.* In: Journal of WSCG, vol. 16. University of West Bohemia, Plzen-Bory, Czech Republic (2008)

[16] Zotti, G., Neumann, A., Purgathofer, W.: *Approximating real-world luminaires with opengl lights.* In: WSCG 2005 Short Paper Proceedings, pp. 49–52. University of West Bohemia, UNION press, Plzen (2005). Available online: `http://www.cg.tuwien.ac.at/research/publications/2005/zotti-2005-lum/`

4. An Extended GPU Radiosity Solver Including Diffuse and Specular Reflectance and Transmission

In this paper we present an extended GPU progressive radiosity solver which integrates ideal diffuse as well as specular transmittance and reflection. The solver is capable to handle multiple specular reflections with correct mirror-object-mirror occlusions. The use of graphics hardware allows to consider attenuation of radiation due to reflections and/or transmissions on a per pixel basis, enabling us to handle multiple specular triangles with different reflection coefficients at once. Alpha masks are used to replace complex geometry in certain cases to reduce computation times. Furthermore, the inclusion of ambient overshooting into the radiosity solver is discussed.

4.1. Introduction

The problem of light interreflection for realistic image synthesis has always been a major issue in the field of computer graphics. Various algorithms to solve the global illumination problem have been proposed over the years. The radiosity method – which is based on the theory of radiative heat transfer – was introduced to the graphics community by Goral et al. [10].

In its original formulation the algorithm does not allow occluders and required time and storage costs of $O(n^2)$ (where n is the number of surfaces) and therefore was impractical for scenes with higher complexity. Cohen and Greenberg [6] introduced the hemicube method to include occluders into the radiosity process. Performance was greatly improved by using progressive refinement [5] or various hierarchical radiosity techniques (e.g. [4, 12]).

Whereas these methods try to reduce the computational complexity, overshooting algorithms have been proposed to speed up the convergence of the radiosity computation itself. In contrast to standard progressive radiosity where shooting is always performed from the patch with the highest unshot radiosity, overshooting methods estimate the radiosity that will be reflected back later on and shoot it together with the actual unshot radiosity. Several overshooting techniques have been proposed, most notable: positive overshooting [25], super-shoot-gather [11] and ambient overshooting [8].

The introduction of programmable graphics hardware has fostered research in hardware accelerated global illumination. Nielsen and Christensen [20] accelerated the hemicube method with the help of hardware texture mapping. Floating point textures to store the result of the radiosity computation where first utilized by Carr et al. [3]. The first radiosity solver which completely works on a GPU was proposed by Coombe et al. [7]. However, the algorithm was restricted to planar quadrilaterals. In [29] we described a GPU radiosity solver for triangular meshes which was based on the work of Coombe et. al.

Furthermore, classical radiosity assumed that all surfaces exhibit ideal diffuse reflection characteristics. To include abitrary reflectance distributions Immel et al. [13] placed an imaginary discretized cube around a vertex to store the incoming light directions. Although this could be implemented on graphics hardware using cube maps, storage requirements to achieve enough accuracy would be too high for more complex scenes. To avoid these memory requirements Rushmeier and Torrance [24] propagate the energy immediately until it reaches a diffuse only surface. They only considered ideal diffuse/specular reflectance and transmittance and no two ideal specular surfaces were allowed to see each other. For planar specular reflecting surfaces they mirror the environment across the surface and propagate the light from the shooter to the *virtual* surfaces. However, instead of reflecting the whole environment, one can simply reflect the shooter around the surface, as noted by Sillion et al. [26]. Li and Yang [17] use a precomputed reflection tree to handle multiple

reflections. The root is the real world and for each mirror (either in the real or a virtual world) a virtual world is constructed and inserted into the reflection tree. The process stops if the estimated form-factor between the real and a virtual world is lower than a certain threshold.

The main contribution of this paper is the description of an extended radiosity solver which is capable of integrating diffuse/specular reflections and transmissions and which performs all calculations on graphics hardware. We propose a unified data structure and rendering path which can handle interreflections and transmissions in arbitrary order. Furthermore we use texture maps with alpha masks to reduce geometric complexity and show that ambient overshooting can easily be included into the process.

The remainder of this paper is structured as follows. In Section 4.2 we will briefly review for the sake of completeness the algorithm for diffuse environments. In Section 4.3 the inclusion of diffuse transmitting surfaces as well as specular reflection and transmission is described. Section 4.3.4 and 4.3.5 describe the inclusion of alpha maps and ambient overshooting. Results are discussed in Section 4.4 and the paper is concluded in Section 4.5. The following nomenclature will be used: The letter ρ will be used for reflectance and the letter τ for transmittance. The subscript d stands for diffuse and s for specular. α will denote the absorptance. To ensure conservation of energy $\rho_d + \rho_s + \tau_d + \tau_s + \alpha$ must sum to unity. Vectors are written in bold face.

4.2. Algorithm Outline

In this section we outline an improved version of the radiosity algorithm, which was proposed in [29], for diffuse reflective surfaces only.

For each triangle two power-of-two 32 bit RGBA floating point textures are maintained. One stores the radiosity and the other the residual energy. The alpha channel is used to mask those texels which are occupied by the triangle. Using a one-fits-all texture size for the radiosity and residual textures, as done in [29], proved suboptimal for scenes which consists of objects with coarse and fine meshes. Coarse meshes got undersampled whereas fine meshes got oversampled. To circumvent this problem it is preferable to assign each object in the scene – based on its coarseness – an individual texture size [1]. All textures are stored in larger lightmaps to reduce texture switching during the radiosity calculation. To allow for mipmapping each lightmap only contains textures of a certain power-of-two size.

After this preprocessing step the radiosity solver starts. At the beginning of each radiosity iteration the next shooter is selected. This is accomplished by constructing a mipmap pyramid for each lightmap and reading back the averaged residual energies.

[1] Varying the texture size over triangles of the same object can lead to interpolation artifacts between boundaries of triangles.

It is important to consider only texels which are occupied by the triangle ($\alpha = 0$) during this calculation. The values are read back and the triangle with the highest unshot energy is the next shooter. During this process an ambient radiosity term (see [5]) can also be calculated with negligible computational cost. This triangle is rendered orthographically into a framebuffer with three color attachments. The color attachments hold the interpolated normals, the world positions and, if the triangle has a texture assigned, a textured version of the triangle. This allows us to incorporate colors on a per-texel basis instead of using a constant color for the whole triangle.

Afterwards a visibility texture from the viewpoint of the shooter (the shooter information is sampled from the afore mentioned three textures) is created by using a stereographic projection (as described in [7, 2, 29]. During the rendering an occlusion query is issued for each triangle. Each triangle that might have received energy (this can be determined with the afore mentioned occlusion query) is rendered orthographically to a framebuffer and the texels are back-projected into the shooters viewpoint. If a certain texel of a receiver is deemed visible from the viewpoint of the shooter, that is, the depth stored in the visibility texture matches the depth of the texel, then the form-factor is evaluated in a fragment shader.

We use the disc approximation to the differential form factor, as proposed by Wallace et al. [27], which is given by

$$F_{dA_i,A_j}(= F_{i,j}) = A_j \sum_{i=1}^{k} \frac{\cos(\phi_i)\cos(\phi_j)}{d^2\pi + \frac{A_j}{k}} \tag{4.1}$$

where A is the area of a surface and k the number of sampling points on A_j. The angles ϕ_i and ϕ_j relate the normal of patch i respectively patch j with the vector joining i and j and d is the distance between the two patches. The new radiosity $\mathbf{B}_{new}$ and residual energy $\mathbf{R}_{new}$ of a texel with color $\mathbf{c}$ is then given by

$$\mathbf{B}_{new} = \mathbf{B}_{old} + \mathbf{E} \cdot F_{i,j} \cdot V_{i,j} \cdot \mathbf{c} \cdot (1 - \rho_d) \tag{4.2}$$

$$\mathbf{R}_{new} = \mathbf{R}_{old} + \mathbf{E} \cdot F_{i,j} \cdot V_{i,j} \cdot \mathbf{c} \cdot \rho_d \tag{4.3}$$

where $\mathbf{E}$ is the emission of the shooting patch and $V_{i,j}$ is either 0 or 1 depending if surface i and j are visible to each other or not. ρ_d is the diffuse reflectance and $\mathbf{c}$ the color of the incoming light. After shooting from each texel of the shooting triangle the residual texture of the shooter is set to zero and the next iteration starts. For a more thorough description see [7] or [29].

4.3. Extensions

In this section we describe the inclusion of diffuse transmittance as well as ideal specular transmittance and reflection into the above outlined GPU radiosity solver. Furthermore the use of masking and ambient overshooting will be discussed.

4.3.1. Diffuse Transmission

For a scene with N surfaces the radiosity of a diffuse transmissive element n is given by

$$\mathbf{B}_n = \mathbf{E}_n + \rho_{n,d} \sum_{m=1}^{N} \mathbf{B}_m F_{n,m} + \tau_{n,d} \sum_{m=1}^{N} \mathbf{B}_m T_{n,m} \tag{4.4}$$

where E_n is the self-emission of n and $T_{n,m}$ is the so called backward diffuse form factor, which denotes the fraction of diffuse energy leaving n from its back side and impinging on surface m. $T_{n,m}$ can be computed the same way as (the forward diffuse form factor) $F_{n,m}$, but this time the hemisphere is placed at the back side of the current shooter. A derivation of Equation 4.4 with respect to surface intensities can be found in [24].

To avoid storing the transmittance part, which is received by an element in a separate texture, the residual texture stores the radiosity sum (cf. Equation 4.4) which is available for diffuse reflectance and transmission. If a diffuse transmissive triangle is selected as next shooter, shooting is performed twice. Once in direction of the normal of the shooter (front side) by multiplying the sampled intensity from the residual texture with $\rho_d/(\rho_d + \tau_d)$ to retrieve the amount available for diffuse reflection and once to the back side by multiplying the intensity with $\tau_d/(\rho_d + \tau_d)$ respectively. Since such triangles can receive energy from both sides, the normals of the receiving triangle t are inverted if $t.\mathbf{normal} \cdot \mathit{shooter}.\mathbf{normal} > 0$ for the calculation of the impinging energy. The lamp shade in Figure 4.6 was rendered with diffuse transmissive triangles.

4.3.2. Specular Reflection

Specular reflections are handled in two steps. In what follows, we will first focus on the view-independent component of the specular reflection and afterwards the view-dependent part will be discussed.

As described by Rushmeier and Torrance [24] a specularly reflecting surface can be treated as a window which restricts the view into the *virtual* world behind the mirror. Therefore the environment across the mirror plane is reflected before calculating the window form factor with the hemicube method. Sillion et al. [26] improved the

concept by reflecting the shooting patch across the mirror surface instead of the whole environment (which is valid if the mirror is planar). We will follow the approach by Sillion et al. and reflect the position and normal of the shooter.

As already mentioned in the introduction there are two ways to include specular reflectance into a radiosity solution. Since the incoming direction must be known for specular reflectance either this information has to be stored in a data structure like the global cube as proposed by Immel et al. [13] or the energy must be propagated immediately. To avoid the memory requirements of the former we use the second approach and only store diffuse intensities in textures, whereas specular intensities are traced instantly until a purely diffuse surface is reached. In case of specular reflectance the radiation B_j arriving at a surface j from a surface i after n specular reflections at mirror surfaces m is given by

$$\mathbf{B}_j = \mathbf{B}_i \cdot (\rho_1 \cdot \rho_2 \cdot ... \cdot \rho_n) \cdot F_{dA_i(m_1,m_2,...,m_n)A_j} \tag{4.5}$$

where $\rho_1 ... \rho_n$ are the specular reflectances of the mirror surfaces. This is actually the diffuse form factor multiplied by the specular reflectances, since the radiation is attenuated by every reflection (see e.g. [19] for a precise description). The nomenclature $i(m)$ – as commonly found in radiative heat transfer literature – designates the mirror image of surface i in mirror m. It is worth mentioning that the law of reciprocity holds also true in case of planar specularly reflecting surfaces (see [19] for a proof).

To handle multiple reflections the algorithm proceeds recursively and uses a reflection tree as data structure. Unlike [17] the reflection tree is calculated dynamically and not in a preprocessing step. The recursive algorithm starts if during rendering of the receivers – as described in Section 4.2 – at least one specularly reflecting triangle is encountered. If there is more than one triangle then they are grouped according to their mirror-plane[2] and for every group a node is inserted into the tree. Figure 4.1 shows a scene with two mirror surfaces and Figure 4.2 the corresponding reflection tree.

A node of the tree stores – beside pointers to the children and parent – the mirror plane, the specular reflecting triangles contained in this plane and the normal and position of the already, around the mirror plane, reflected shooter as well as a list of triangles which received light. For each node in the reflection tree the following steps are performed.

First, an appropriate projection for rendering of the visibility texture from the viewpoint of the reflected shooter is constructed. Instead of using a hemispherical projection a perspective projection, where the eye is located at the shooter position and

[2] All surfaces which lie in the same plane can be handled at once since they have the same virtual shooter position.

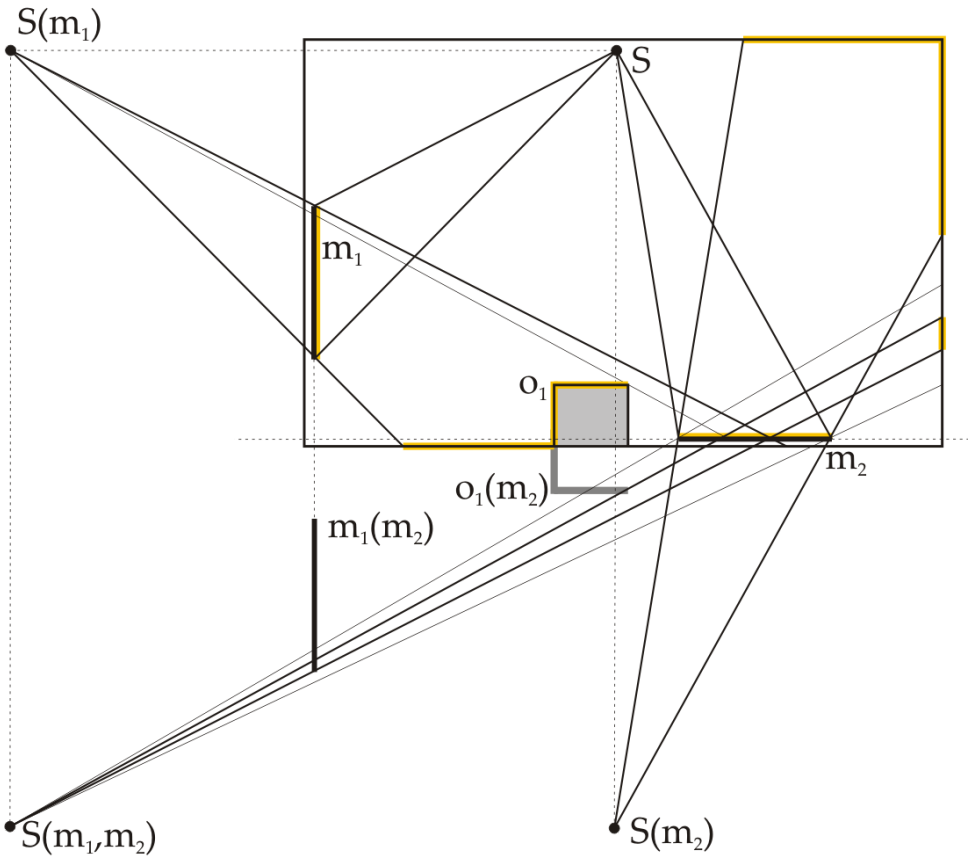


Figure 4.1.: A room with two mirrors m_1 and m_2. The shooter is located at position S. For the second order reflection from point $S(m_1, m_2)$ mirror m_2 is partially occluded by o_1 (for a better overview the room itself is not considered as an occluder). Only surfaces which received light during shooting from $S(m_1)$ may occlude m_2. Furthermore m_2 is only partially visible through m_1.

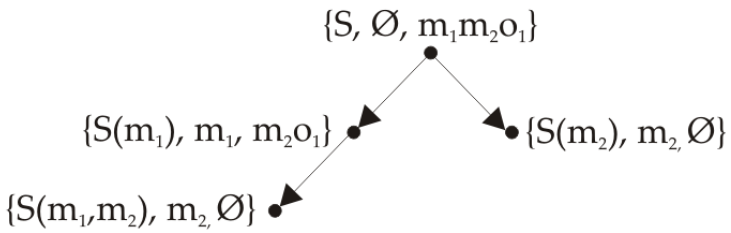


Figure 4.2.: The corresponding reflection tree for the example in Figure 4.1. For each node the shooter position, the reflecting surfaces and the possible occluders are given.

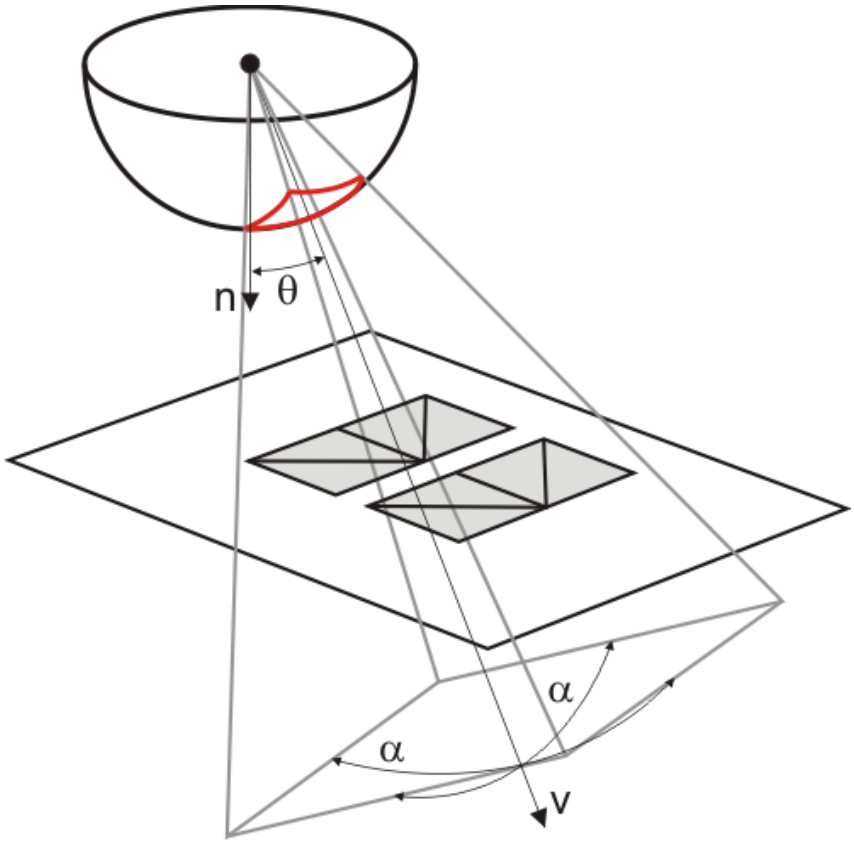


Figure 4.3.: The solid angle subtended by the view frustum is used to calculate an estimate of the unoccluded form factor. The field-of-view angle is chosen in a way such that all reflecting triangles (shown in gray) within the same plane are inside the frustum.

is pointing toward the barycenter of the specular triangles, is used. The field of view angle is set to the minimal angle such that all triangles are projected onto the image plane. This has the advantage that the world seen by the mirror occupies a fairly large portion of the visibility texture. In case of a hemispherical projection the areas can get very small, especially at the perimeter or after multiple reflections.

It is clear that the energy transferred through a sequence of reflections decreases steadily. To avoid spending computation time on nodes which would only add insignificant detail to the final solution an upper bound of the form factor for the current node is estimated. In the work of Hanrahan et al. [12] the unoccluded form factor is estimated by calculating the solid angle subtended by a sphere enclosing the area of the receiver, whereas in [27] a disc with cross sectional area equal to the area of the receiver is used. In our case we do not know in advance which surfaces will receive energy and aside from that the receivers are not handled separately (each node can distribute light to several surfaces). Therefore the solid angle subtended by the view frustum of the above defined projection is used. Thus, in case of a square frustum the form factor estimate F_e is given by

$$F_e = \frac{\cos(\Theta)}{\pi}(4 \cdot \arcsin(\sin^2(\frac{\alpha}{2})) \tag{4.6}$$

where α is the dihedral angle between opposite faces of the view frustum, or to put otherwise, α corresponds to the field-of-view angle. Θ is the angle between the normal of the shooter patch and the view direction. Figure 4.3 depicts the concept graphically. If this estimate is below a certain threshold the recursion is stopped at this point. Otherwise the visibility texture is constructed.

In general, the stencil buffer is used to mask those areas which are visible through the mirrors. If multiple specular reflections and mirror-object-mirror occlusions are not considered it is enough to render the mirror triangles with the stencil test enabled. Otherwise the problem is more involved, since occlusions can occur or a mirror is only partially visible through another mirror (e.g. in Figure 4.1 mirror m_2 is only partially visible through m_1).

In this case the stencil operation is set to increase, and the mirror triangles of the current node are rendered. Then all mirror triangles of all preceding nodes are rendered, whereby in each step from a node n_{child} to n_{parent} upwards in the tree the current modelview matrix is post-multiplied with the reflection matrix stored in the node n_{child}, to reflect the triangles to the appropriate position. Since reflection coefficients and colors can vary from triangle to triangle (or over the surface of a single triangle) multiplicative blending is used to gain the attenuation factors and colors on a per-texel basis. To account for mirror-object-mirror occlusions all possible occluder surfaces contained in preceding nodes are also rendered[3]. Because possible occluder triangles which are behind the mirrors after reflection actually do not occlude the view onto the mirror, the mirror plane of n_{child} is used as clipping plane for the possible occluders in the node n_{parent} to ensure correct visibility. For example, in Figure 4.1 the surface o_1 occludes mirror m_2 and must therefore be reflected at m_2. Furthermore, since a parent node can contain occluder triangles which are also part of the mirror triangles of the actual node, a small offset has to be added to the mirror plane to prevent rendering of these triangles. E.g. when processing node $\{S(m_2), m_2\}$, a small offset is added to the mirror plane defined by m_2 to prevent rendering of m_2 which is part of the occluder triangles of the root node. Rendering of the occluders resets the stencil value to zero at the corresponding positions.

In the next step all potential receivers are rendered only to those areas where the value in the stencil buffer equals the current recursion depth. This is necessary since a mirror need not be contained as a whole in another's mirror visibility window as mentioned above. To clip away the geometry behind the mirror plane either an additional clipping plane has to be specified or the near view plane must be aligned with the mirror plane using the oblique near-plane clipping technique as described

[3] Occlusions which occur between a mirror and a diffuse receiver are accounted for with the visibility texture. Figure 4.5 shows an example where a mirror-object-mirror occlusion occurs.

in [16]. The latter has the advantage that there is no additional clipping plane necessary, but it reduces the depth buffer precision depending on the alignment. We therefore preferred the former technique.

The perspective projection also increases the amount of geometry which can instantly be culled away. Before rendering the individual triangles view frustum culling with occlusion queries is performed, since objects inside the view frustum need not necessarily be seen by a mirror (that is, would not pass the stencil test). All triangles of objects which passed the occlusion query are rendered into the visibility texture. Again, an occlusion query is issued for each triangle as described in Section 4.2.

If a diffusely reflecting or transmissive surface is deemed visible (the depth stored in the visibility buffer equals the depth of the currently examined fragment) then the radiosity and residual texture are updated with

$$\mathbf{B}_{new} = \mathbf{B}_{old} + \mathbf{E} \cdot F_{i(m_1,..,m_n)j} \cdot V_{i,j} \cdot \mathbf{c} \cdot \mathbf{c}^* \cdot \rho_s^* \cdot \alpha \tag{4.7}$$

$$\mathbf{R}_{new} = \mathbf{R}_{old} + \mathbf{E} \cdot F_{i(m_1,..,m_n)j} \cdot V_{i,j} \cdot \mathbf{c} \cdot \mathbf{c}^* \cdot \rho_s^* \cdot \lambda \tag{4.8}$$

where $\lambda = (\rho_d + \tau_d)$ and ρ_s^* is the multiplied specular reflectance sampled from the visibility buffer and $\mathbf{c}^*$ are the multiplied color values respectively. All of these triangles are inserted into the possible occluder list of the current node. It is clear that only triangles which received light can occlude a specular reflecting surface in the next recursion step. If specular surfaces are encountered, these are grouped again according to their mirror plane and a child node for each group is inserted into the reflection tree. Listing 4.1 and 4.2 summarize this steps.

```
specularRecursion(Node n) {
 set perspective projection;
 f = estimate unoccluded form factor;
 if (f < threshold) return;
 renderVisibilityTexture(&n);
 for (each receiver r) {
  update radiosity and residual texture of r;
  add r to possible occluder list of n;
  if (r == specular reflective)
   add r to specular surface list L;
 }
 group all surfaces s in L;
 for (each group g) {
  add a child n_c to n;
  specularRecursion(n_c);
 }
}
```

Listing 4.1: Pseudocode for handling specular reflections

```
renderVisibilityTexture(Node *n) {
  stencil operation = increase;
  render specular surfaces of n;
  enable multiplicative blending;
  push modelview matrix mv;
  Node *f = n;
  while (f->parent != NULL) {
    multiply mv with f->reflectionMatrix;
    f = f->parent;
    render specular surfaces of f;
  }
  pop mv;
  disable blending;
  stencil operation = zero;
  push mv;
  f = n;
  while (f->parent != NULL) {
    setClippingPlane(f->mirrorPlane);
    multiply mv with f->reflectionMatrix;
    f = f->parent;
    render occluder surfaces of f;
  }
  pop mv;
  stencil operation = keep;
  renderScene();
}
```

Listing 4.2: Pseudocode for rendering the visibility texture.

To render the view-dependent mirror reflections in real time, a recursive texture-based approach is used. Planar recursive reflections have usually been rendered either by using texture mapping [18] or the stencil buffer. The basic idea is to render the reflected scene into a texture and apply this map onto the reflective polygon. Yet, as high resolutions may be necessary to capture reflections accurately and a texture map is assigned to each polygon, memory requirements may become quite large. Nielsen and Christensen [21] overcome these memory requirements by immediately discontinuing rendering of the scene if a reflecting polygon is met. In this case the corresponding reflection map is created, applied to the reflecting polygon and then the rendering task is continued. This process can be repeated recursively.

In this work we follow this approach since texture memory requirements are already very large for storing the radiosity information. However, instead of calculating the reflection map separately for each reflector, we group them again in regard to their mirror plane. This way multiple triangles can be handled at once. The same reflection tree data structure as above is used. Naturally this time the eye position is reflected around the mirror plane instead of the shooter position. The recursion starts if specular reflecting triangles are visible from the eye point and the recursion stops if either a maximum number of recursions is reached or the mirror size in the final image is below a certain threshold. For each recursion depth one reflection map

with its corresponding depth buffer is created. This is necessary since the contents of a reflection map of a certain recursion level must not be discarded until all child nodes are processed. If such a reflection map for a specific level has to be created the corresponding texture map and depth buffer are bound to a framebuffer object and the scene is rendered from the reflected eye position.

4.3.3. Specular Transmission

Inclusion of specular transmission is more demanding since refraction can occur. Rushmeier and Torrance [24] neglected the effect of refraction at a transmissive surface. In this case the shooter can remain at its position and the transmissive surface acts as window restricting the view of the environment. Leiss et al. [15] considered refractions only if the caused bending is small. To determine the receiving vertex, rays are shot through the transmitting patch and the hit is assigned to the nearest vertex. Since in presence of refraction it is not possible to calculate a single virtual shooter position valid for a whole triangle (in fact the position would describe a caustic surface, as discussed in [9]). Therefore the effect of refraction is neglected in the current implementation, which is valid if the interface is a thin transmitting plate, such as a window [24].

Transmissive surfaces can easily be included into the reflection tree algorithm described in Section 4.3.2. Instead of reflecting the shooter position, the shooter stays where it is and instead of the ρ_s, τ_s has to be used. Furthermore, the modelview matrix for reflecting surfaces is not modified at a transmissive node. Using the same rendering path and data structure has the advantage that reflective and transmissive surface can interact with each other in any arbitrary way. If a surface is specularly reflecting and transmitting two individual nodes are inserted into the reflection tree. Additionally transmitting surfaces are inserted into a BSP-Tree to guarantee correct visibility.

4.3.4. Masking

For masking texture maps with an alpha channel which determine which parts of a triangle are fully transparent ($\alpha = 0$) and which ones are opaque ($\alpha > 0$) are used. The main objective is to speed up the radiosity process by replacing complex geometry with a simple proxy object. They can be used for objects which contain fine geometric details (e.g. fences, grids) and would therefore require complex triangulations. For example, the jalousies in Figure 4.6 were rendered with such an alpha mask.

To include such masks in the above algorithm some minor modifications are necessary. First, texels with an alpha value of zero do not contribute to the area of the triangle and therefore the area of the triangle must be corrected with respect to those texels. Therefore a triangle with a mask is rendered twice during initialization

of the lightmaps. Once the complete number of occupied texels n_c is retrieved from an occlusion query and once only texels which pass an alpha test, which discards all texels with an alpha value of zero, are counted (n_o). The revised area A_r is then given by the multiplication of the original triangle area A with n_o/n_c. The area of a single texel is thus estimated with A_r/n_o (if a triangle has no alpha mask the texel area is given by A/n_c). Second, during rendering of the visibility texture texels with an alpha value of zero have to be discarded to ensure correct visibility. Third, to guarantee correct back-to-front rendering each triangle with an alpha mask is inserted into a BSP-Tree.

Finally, it should be noted that the resolution of the radiosity/residual texture of a triangle should match the resolution of the texture used as mask. Otherwise artifacts will appear since the illumination of the triangle does not correctly correspond to its visibility information.

4.3.5. Overshooting

Overshooting techniques aim to speed up the convergence of the radiosity computation. From the techniques mentioned in the introduction ambient overshooting is the most applicable one for our GPU radiosity solver. The former two would require patch-to-patch form factors by the time of shooting and – even less advantageous – additional storage to record the pre-shot radiosity from a patch i to a patch j. On the other hand, ambient overshooting utilizes the ambient term, which in our case is already calculated during the next shooter selection and for display purposes. Furthermore, no additional memory is required and, as shown by Gortler et al. [11], ambient overshooting and super-shot-gather performed equally well in terms of convergence.

To include ambient overshooting in the proposed algorithm two small changes are required. First, instead of using the patch which maximizes $\Delta B_i A_i$ for shooting as in traditional progressive radiosity, the patch which maximizes $(\Delta B_i + \Delta B'_i)A_i$, where $\Delta B'_i$, according to Feda and Purgathofer [8], is given by

$$\Delta B'_i = min(\rho_d^i \cdot ambient, \sum_{j=1}^{n} \frac{\Delta B_j \cdot A_j}{A_i} - \Delta B_i) \tag{4.9}$$

This equation ensures that the ambient term never becomes negative (if it would, the solution would not converge, as shown by Feda and Purgathofer [8]). Second, after shooting the unshot radiosity, patch i must be set to $-\Delta B'_i$. Since floating point textures are used storage of negative values is no problem. This negative value should tend back towards zero if the estimate was good. However, if $\Delta B'_i$ was overestimated a negative amount must be shot back into the environment. Ambient overshooting led to a decrease in rendering time of about 10 to 40 percent, depending on scene

Scene	n_t	n_{texels} [sec]	n_s [sec]	t_r^{sp} [sec]	t_{spec} [sec]	t_r^{sp+ao} [%]	t_r^{hc} [%]	t_r^{hc+ao} [%]
Cornell box$_{high}$	259	126782	361	4.97	0	-38.81	0.01	-33.72
Cornell box$_{low}$		29666	89	2.39	0	-37.73	3.40	-36.75
living room$_{high}$	3821	2952096	4106	192.94	41.73	-37.86	-5.64	-37.18
living room$_{low}$		1401806	1137	60.32	13.22	-12.83	-13.14	-24.89
bedroom$_{high}$	12355	2054225	1049	216.62	59.54	-22.80	8,04	-15.03
bedroom$_{low}$		381230	269	67.50	18.19	-19.12	6.07	-11.74
teahouse$_{high}$	4442	2941550	3621	415.59	212.95	-32.24	21.92	-10.21
teahouse$_{low}$		690870	916	127.75	65.00	-32.16	-22.14	-42.01

Table 4.1.: Performance of the presented algorithm. The columns show the number of triangles n_t, the number of texels n_{texels} along with the number of shots n_s and the time needed by the radiosity solver t_r to reach 90% convergence. In addition the increase/decrease in rendering time with ambient overshooting (ao) for sterographic projection (sp) and the hemicube (hc) is given.

geometry and reflectivity (see Table 4.1).

4.4. Results and Discussion

We implemented the described methods with C++, OpenGL and the Cg shading language. Table 4.1 shows information about the examples which have been used in this paper. All time measurements were performed on an Intel Core2 CPU with 2.13 GHz, 3.5 GB RAM with a Geforce 8800GTS with 640MB DDR3 Ram. Each scene was evaluated twice, once with higher resolution textures (high) and once with reduced resolution (low). A 512x512 texture was used for the hemicube and a 1024x1024 visibility texture for the stereographic projection. Shooting was performed each time from the third mipmap level. By always shooting from the same mipmap level regardless of the texture size of a current shooter and by assigning lower resolution maps to triangles with smaller area has the benefit that the number of shootings corresponds naturally to the size of the triangle. Furthermore, lower resolution textures reduce the number of shootings per map, simultaneously the energy per shooting iteration is increased accordingly, leading to an overall decrease in shooting steps which are necessary to reach the same convergence.

In Figure 4.5 one diffuse area light source is positioned slightly in back of the woman, whereas Figure 4.6 uses a goniometric light source for the floor lamp and diffuse area light for a streetlamp outside the room. Goniometric lightsources are rendered with bicubic filtered light distribution textures, as described in [28]. The scene in Figure 4.7 has two diffuse area light sources placed at the ceiling and outside of the room. To map the high dynamic range information to the output range of [0..1] a simple exposure function was used for Figure 4.5 and 4.7. In case of Figure 4.6 the global tone mapping operator from Reinhard [22] was applied.

To compare the quality of the results achieved with the stereographic projection,

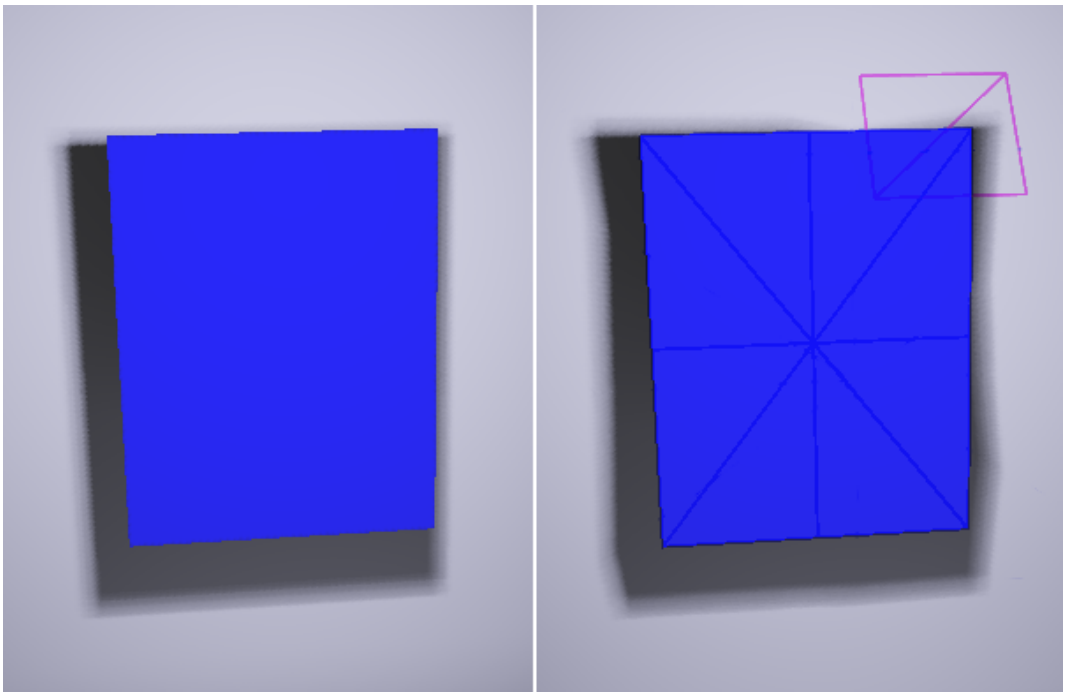


Figure 4.4.: Comparison between the hemicube (left) and stereographic projection (right). Since the triangulation of the occluder is too course the shadow boundary is distorted in case of the stereographic projection.

we also implemented the hemicube method for rendering of the visibility texture. Although the stereographic projection requires rendering of the scene only once, instead of five times when using a hemicube, it has some serious drawbacks due to the fact that only vertices are affected by the projection. This can result in wrong visibility information (e.g. areas are not recognized as visible) especially if the triangles are large or the light sources are near to an object. Figure 4.4 compares the results gained with the hemicube and the stereographic projection by means of a simple test scene. This is the reason why in some cases the hemicube performed better because less shootings were necessary to reach 90% convergence (e.g. in the teahouse scene). A possibility to improve the result would be to use spherical rasterization as described in [14]. As they note, a pixel belongs to the spherical projection if it is above all three planes defined by the edges of the triangle and the origin. However, Kautz et al. [14] only used a low resolution visibility mask, so they could speed up the process by precomputing a lookup table of bitmasks for a discrete set of planes. In our case the spherical rasterization would be too time consuming. In this sense a programmable rasterization stage would be desirable.

It is worth mentioning that rendering the receivers is the most time consuming function, since it is invoked for each visible receiver for each shooting of light. Performance analysis showed that the call to cgGLSetStateMatrixParameter – to setup the camera for orthographic rendering of the triangles – uses most of the time. The

problem seems to be an internal glGetDoublev call which stalls the pipeline. We found that the best option is to access the modelview projection matrix directly in the shader via the glstate directive from Cg.

In our discussion we implicitly assumed that all surfaces are gray emitters where neither ρ nor τ depends on wavelength (or incoming direction). However, it is clear that the method can easily be extended to allow for separate radiative properties for each color channel.

It must be pointed out that grouping triangles according to common planes can cause some trouble if the field-of-view angle for the perspective projection approaches 180 degree. Imagine a hallway with several mirrors hanging on the same wall. In such a scenario all mirrors would be in the same plane, spanning a wide distance. If this is the case it is preferable to split such a node into several ones with smaller viewing angles, otherwise artifacts will appear due to distorted visibility information.

4.5. Conclusion

An extended radiosity solver which accounts for both diffuse and specular reflection and transmission has been presented. Whereas diffuse intensities are stored in textures, specular intensities are traced immediately until they reach a diffuse only surface. The solver is capable to handle multiple specular reflections with correct mirror-object-mirror occlusions. Transmissions are handled within the same recursive rendering path and data structure. To accomplish this, a data structure called a reflection tree was used. The use of hardware texture mapping easily allows to consider attenuation and color bleeding on a per-texel basis.

Furthermore the inclusion of ambient overshooting and masking, to speed up the radiosity process, was discussed. The former tries to improve the convergence of the radiosity computation, whereas the latter tries to reduce the geometric complexity. Images of several scenes showed results which were achieved with our implementation.

The stereographic projection can lead to some severe artifacts if the triangulation of the scene is not fine enough. The hemicube circumvents these problems at cost of higher computation times. In this sense a fast heuristic to decide if the error by the hemispherical projection is negligible or not would be useful.

To consider participating media Rushmeier and Torrance [23] used the so-called zonal method. By altering the form factors of the zonal method it can also be extended to specular surfaces as described in [1]. We consider to include participating media into the GPU radiosity solver since the zonal method may be well adoptable to 3D textures.

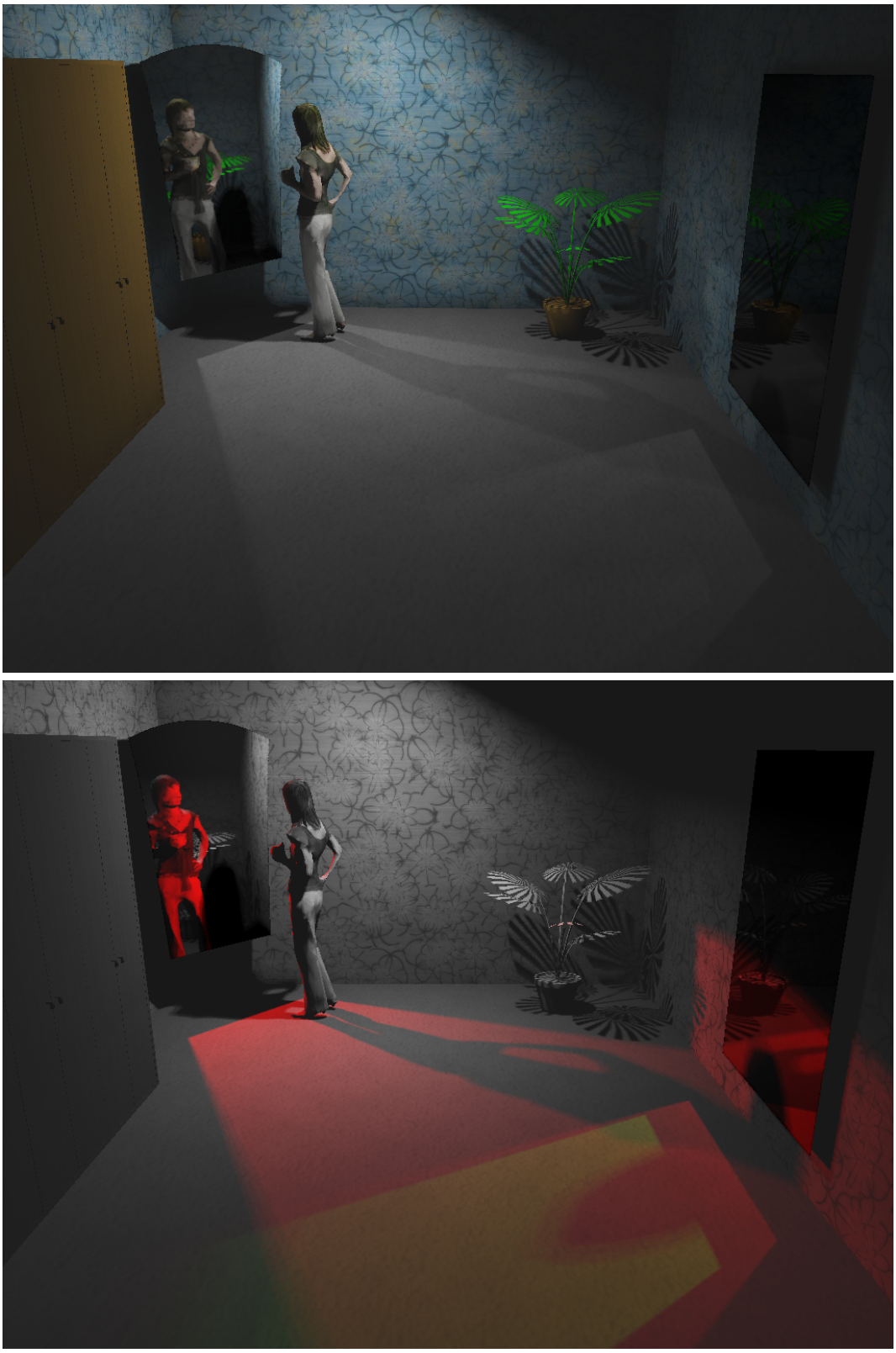


Figure 4.5.: A room with two mirrors. One lightsource is pointing towards the mirror in the corner. The woman partially occludes the light path from the mirror in the corner to the other one, resulting in a partly occluded 2nd order reflection. Furthermore, only the part from the wall mounted mirror which is seen through the corner mirror casts a 2nd order reflection. The image on the bottom shows the 1st and 2nd order reflections in red and green respectively for the shooting steps from the light source.

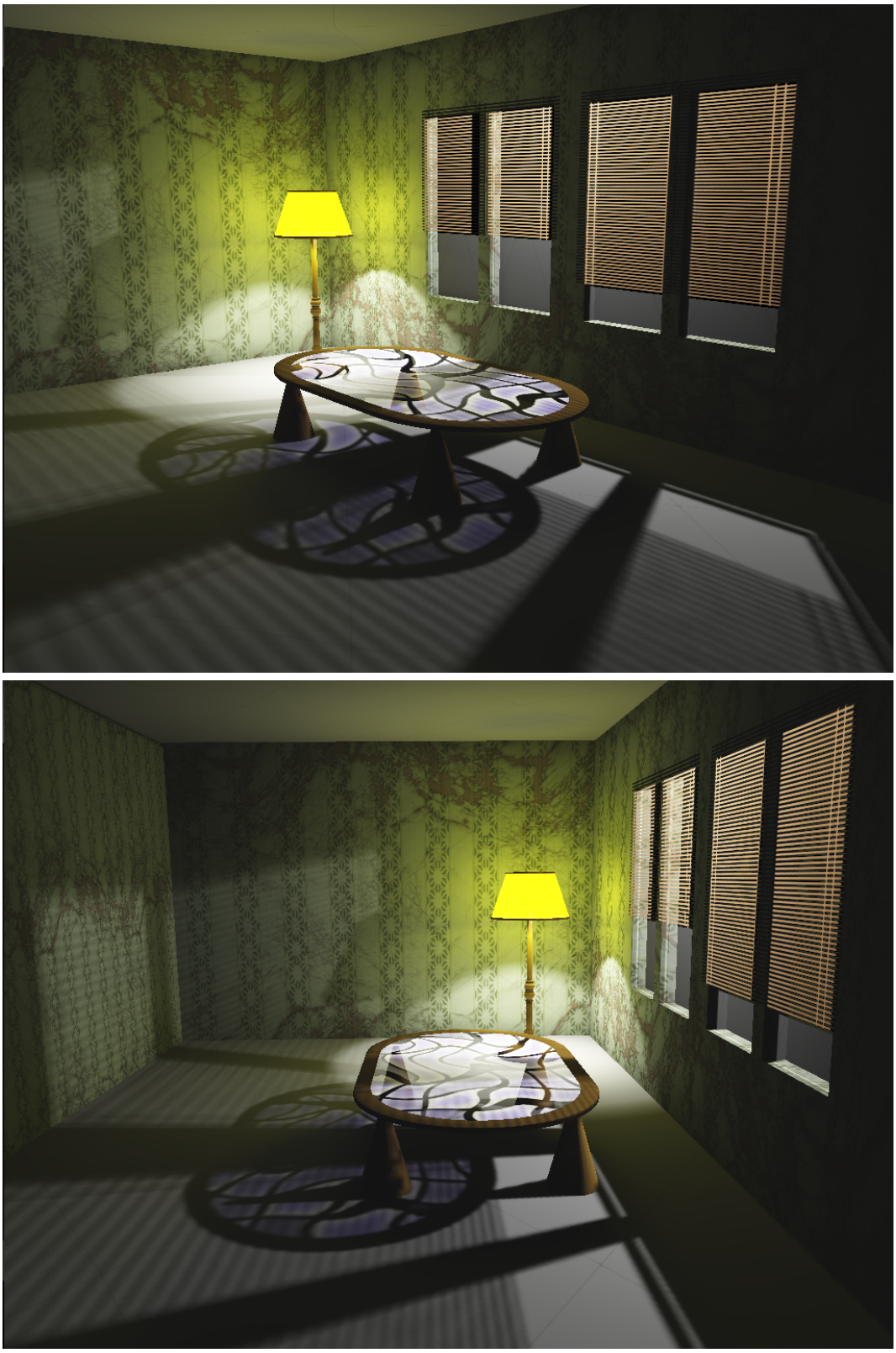


Figure 4.6.: For the lamp shade diffuse transmitting surfaces where used. One goniometric light source was placed inside the lamp and one standard diffuse area light outside the room. The window and glas table are specularly transmitting. The colors for the lighting calculations are sampled from the texture. The jalousies where modeled as simple planes with an alpha mask.

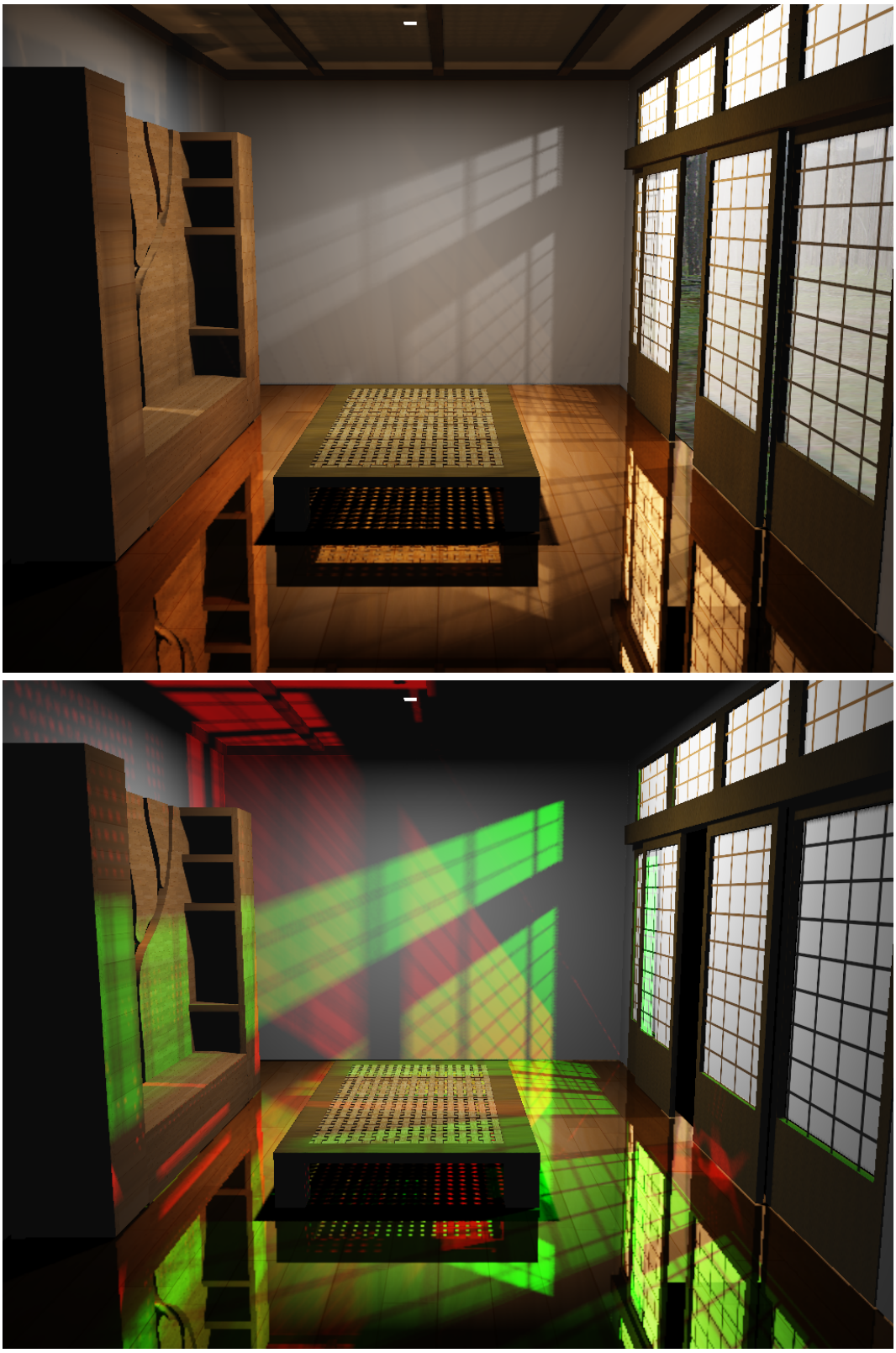


Figure 4.7.: A japanese teahouse with one light source located at the ceiling and another higher intensity source outside. The doors have specular transmissive windows with a high absorptance rate. The floor is specular reflective and behind the wallsystem a highly reflective mirror was placed. Alpha masking was used for the grids on the doors and the table board . The image on the bottom shows transmitted light in green and reflected light in red for the shooting steps of the lightsource outside.

4.6. Bibliography

[1] Arquès, D., Michelin, S., Piranda, B.: *Extending the zonal method to specular surfaces.* Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization (1998)

[2] Barsi, A., Jakab, G.: *Stream processing in global illumination.* Proceedings of 8th Central European Seminar on Computer Graphics (2004)

[3] Carr, N.A., Hall, J.D., Hart, J.C.: *Gpu algorithms for radiosity and subsurface scattering.* In: HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 51–59. Eurographics Association, Aire-la-Ville, Switzerland (2003)

[4] Cohen, M., Greenberg, D., Immel, D., Brock, P.: *An efficient radiosity approach for realistic image synthesis.* Computer Graphics and Applications **6**(3), 26–35 (1986)

[5] Cohen, M.F., Chen, S.E., Wallace, J.R., Greenberg, D.P.: *A progressive refinement approach to fast radiosity image generation.* In: SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques, pp. 75–84. ACM Press, New York, NY, USA (1988)

[6] Cohen, M.F., Greenberg, D.P.: *The hemi-cube: a radiosity solution for complex environments.* In: SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 31–40. ACM Press, New York, NY, USA (1985)

[7] Coombe, G., Harris, M., Lastra, A.: *Radiosity on graphics hardware.* Technical report, Univ. of North Carolina, UNC TR03-020 (2003)

[8] Feda, M., Purgathofer, W.: *Accelerating radiosity by overshooting.* In: Third Eurographics Workshop on Rendering, pp. 21–32 (1992)

[9] Glaeser, G., Schroecker, H.P.: *Reflections on refractions.* In: Journal for Geometry and Graphics, vol. 4, pp. 1–18 (2000)

[10] Goral, C.M., Torrance, K.E., Greenberg, D.P., Battaile, B.: *Modeling the interaction of light between diffuse surfaces.* In: SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 213–222. ACM Press, New York, NY, USA (1984)

[11] Gortler, S., Cohen, M., Slusallek, P.: *Radiosity and relaxation methods.* In: Computer Graphics and Applications, IEEE, vol. 14, pp. 48–58 (1994)

[12] Hanrahan, P., Salzman, D., Aupperle, L.: *A rapid hierarchical radiosity algorithm.* SIGGRAPH Comput. Graph. **25**(4), 197–206 (1991)

[13] Immel, D.S., Cohen, M.F., Greenberg, D.P.: *A radiosity method for non-diffuse environments.* SIGGRAPH Comput. Graph. **20**(4), 133–142 (1986)

[14] Kautz, J., Lehtinen, J., Aila, T.: *Hemispherical rasterization for self-shadowing of dynamic objects.* In: Proceedings of Eurographics Symposium on Rendering 2004, pp. 179–184. Eurographics Association (2004)

[15] Leiss, T., Ferschin, P., Purgathofer, W.: *Radiosity with textures, specular reflection and transmission.* In: Proceedings of the Fourteenth Spring Conference on Computer Graphics, pp. 103–111 (1998)

[16] Lengyel, E.: *Oblique view frustums for mirrors and portals.* In: Game Programming Gems 5. Charles River Media (2005)

[17] Li, S.Y., Yang, S.N.: *Radiosity for scenes with many mirror reflections.* In: The Visual Computer, vol. 16, pp. 481–500 (2000)

[18] McReynolds, T., Blythe, D., Grantham, B., Kilgard, M.J.: *Advanced graphics programming techniques using opengl.* SIGGRAPH 1999 Course (1999)

[19] Modest, M.F.: *Radiative Heat Transfer*, 2nd edn. Academic Press (2003)

[20] Nielsen, K.H., Christensen, N.J.: *Fast texture-based form factor calculations for radiosity using graphics hardware.* J. Graph. Tools **6**(4), 1–12 (2002)

[21] Nielsen, K.H., Christensen, N.J.: *Real-time recursive specular reflections on planar and curved surfaces using graphics hardware.* pp. 91–98 (2002)

[22] Reinhard, E.: *Parameter estimation for photographic tone reproduction.* J. Graph. Tools **7**(1), 45–52 (2002)

[23] Rushmeier, H.E., Torrance, K.E.: *The zonal method for calculating light intensities in the presence of a participating medium.* In: SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pp. 293–302. ACM Press, New York, NY, USA (1987)

[24] Rushmeier, H.E., Torrance, K.E.: *Extending the radiosity method to include specularly reflecting and translucent materials.* ACM Trans. Graph. **9**(1), 1–27 (1990)

[25] Shao, M.Z., Badler, N.I.: *A gathering and shooting progressive refinement radiosity method.* Technical Report MS-CIS-93-03, University of Pennsylvania (1993)

[26] Sillion, F.X., Arvo, J.R., Westin, S.H., Greenberg, D.P.: *A global illumination solution for general reflectance distributions.* SIGGRAPH Comput. Graph. **25**(4), 187–196 (1991)

[27] Wallace, J.R., Elmquist, K.A., Haines, E.A.: *A ray tracing algorithm for progressive radiosity.* In: SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques, pp. 315–324. ACM, New York, NY, USA (1989)

[28] Wallner, G.: *Geometry of arbitrary light distributions.* In: Proceedings of the 13th International Conference on Geometry and Graphics (2008)

[29] Wallner, G.: *GPU radiosity for triangular meshes with support of normal mapping and arbitrary light distributions.* In: Journal of WSCG, vol. 16 (2008)

A. Force Directed Embedding of Hierarchical Cluster Graphs

Cluster graphs are a valuable concept to visualize structured relational information. Hierarchical cluster graphs impose further levels of granularity which may be controlled by the user. In this paper we present a force directed layout adjustment algorithm for hierarchical cluster graphs. Clusters and cluster hierarchies respectively, can be dynamically closed or opened which is vital to selectively reduce the information presented by large graphs. In our case such an operation only rearranges the graph locally, therefore preserving the mental map of the user. We also present results achieved with our algorithm in the domain of semantic net exploration.

A.1. Introduction

Force directed placement (FDP) algorithms evolved from a VLSI technique originally described by Quinn and Breuer [19]. Eades [3] based his work on the afore mentioned technique and adopted it for the drawing of undirected graphs. He modeled edges of the graph as springs, but his force model does not reflect Hooke's law, unlike Kamada and Kawai [10] who solved a set of linear equations. Fruchterman and Reingold [7] improved performance of computation by changing the force model. Since then many different force models have been developed.

With increasing amount of data, graphs tend to become more cluttered and visually incomprehensible. Clustered graphs are a way to bring more structure on top of the classical graph model. Different algorithms for the drawing of clustergraphs have been proposed since the clustered graph model was introduced by Eades et al. [4]. As mentioned by Brockenauer and Cornelsen [1] a simple method would be to add dummy attractors for each cluster and connect every node of that cluster with the attractor. Nodes of the same cluster will therefore be closer together. Huang and Eades [9] introduced another approach which uses three different spring forces. For an edge which connects two vertices belonging to different clusters another spring force is used as for an edge which connects vertices of the same cluster. The third spring force is used for edges between a virtual vertex (dummy attractor) and a vertex. A similar approach was taken by Wang and Miyamoto [21], which used the concept of a meta-graph instead of dummy vertices. Related to this problem, is the issue of how to remove node overlapping for non-point vertices. This issue was first addressed by Misue et al. [14]. Recently, Li et al. [12] introduced a FDP algorithm which uses a dynamic natural length and preserves the mental map ([14]). It was proven by Nagamochi and Kuroya [16] that transforming a layout with overlapping nodes into a minimum area layout without node overlapping is NP-hard.

The problem of how nodes should be grouped into clusters also spawned widespread research interest. A good overview of clustering algorithms can be found in [8]. In recent years clustering approaches with user involvement became popular. For example, Nascimento and Eades [17] presented a method where users can give "hints" that help the computer to find better solutions.

The algorithm presented in this paper is a layout adjustment algorithm, since it is dependent on an already existing initial layout of the graph. This work is based on the work of Wang and Miyamoto [21] and Li et al. [12] extending it to support subcluster hierarchies. The remainder of this paper is structured as follows. Section A.2 reviews in short the definitions which are used throughout this work. In Section A.3 our algorithm is described in detail. Results obtained with our method are presented in Section A.4. Before we conclude the paper in Section A.6 topics for future work are presented in Section A.5. Section A.3.5 outlines in short our automatic clustering algorithm.

A.2. Definitions

A graph is a tuple $G = (V, E)$ where V is a finite set of nodes and E is a finite set of edges. An edge $e = (v_1, v_2)$ $v_1, v_2 \in V$ connects the two vertices v_1 and v_2.

Clustergraphs are an extension of graphs, which exhibit a hierarchical structure of clusters, in which the nodes of the graph are organized. A clustered graph (or clustergraph) is a graph with a partition $(C_1, ..., C_k)$ on the vertex set. C_i $(i = 1...k)$ are called cluster. A (k-way) partition of a set C is a family of subsets $(C_1, ..., C_k)$ with $\cup_{i=1}^{k} C_i = C$ and $C_i \cap C_j = 0$ for $i \neq j$ [1]. A hierarchical clustergraph G_c is a tuple $G_c = (G, T)$ where T is a rooted tree, whose leaves are exactly the nodes of G (according to [4]) . A cluster C_i is a subcluster of a cluster C_j, if and only if C_j is an ancestor of C_i in T.

For the following discussion we define a meta-graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a finite set of meta-nodes and $\mathcal{E}$ is a finite set of meta-edges. A meta-node ν_i consists of all vertices of cluster C_i. Because we would like that clusters containing nodes connected with each other should be close together – to keep edges in the final embedding short – a meta-edge $\varepsilon = (\nu_i, \nu_j)$ connects the two meta-node ν_i and ν_j if there exists an edge $e = (v_i, v_j)$ $i \neq j$ with $v_i \in C_i$ and $v_j \in C_j$. Each two meta-nodes ν_i and ν_j $i \neq j$ are connected by at most one meta-edge.

A.3. Algorithm

First, the clustering of graph G has to be defined. This can be done either automatically, semi-supervised or manually. For example the popular k-means clustering algorithm [13] classifies n nodes into k clusters by assigning each node to the cluster whose average value on a set of p variables is nearest to it by some distance measure on that set. A semi-supervised variant of the k-means algorithm has been published by Wagstaff et al. [20]. A brief survey of clustering methods can, for example, be found in [18]. We used our clustering algorithm outlined in Section A.3.5.

Second, the graph G is layouted without considering cluster information. This can be done by classical force directed layout algorithms ([7, 3, 2]) or of course with any other graph drawing algorithm. This step yields an initial graph layout. From this layout the metagraph $\mathcal{G}$ has to be derived (see Section A.3.1). Finally, the metagraph is layouted with the algorithm described in Section A.3.2.

A.3.1. Deriving the metagraph

The meta-graph $\mathcal{G}$ is derived from G_c by creating a meta-node for each cluster. A meta-node stores all nodes which belong to the cluster. Furthermore, the superordinate meta-node and a list of subordinate meta-nodes is saved, yielding a tree structure. For faster access during the layout process the “branching point" is also

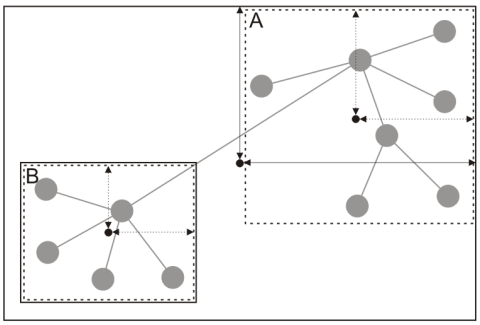


Figure A.1.: Two meta-nodes, whereas B is subordinate to A. Since B does not have any subordinate meta-nodes the inner bounding box equals the outer. The small dots indicate the centers of the boxes and the arrows the half width and height respectively.

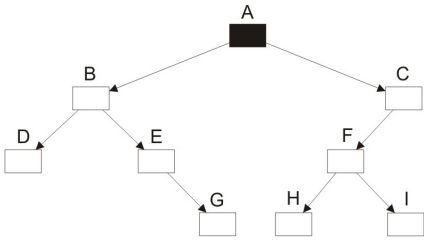


Figure A.2.: A tree of a graph which consists of 9 meta-nodes, whereas B is branching point for D, E and G. Node F is branching point for H and I and the root A is branching point for B, C and F.

stored. In addition two bounding boxes are stored. In the reminder of this paper we will refer to them as inner bounding box and outer bounding box.

The inner bounding box encloses only the nodes of the current meta-node. The outer bounding box encloses the nodes of the meta-node itself and of all subordinate meta-nodes. If a meta-node is a leaf then the outer bounding box equals the inner bounding box. A bounding box is defined by its center and the half length in x and y direction. Since each cluster is surrounded with a slightly inflated convex hull to allow for better distinction, a fixed value is added to the half length to accommodate that hull. The size of the outer bounding box can simply be determined by traversing the tree. Figure A.1 shows the bounding boxes with their center and half lengths.

The “branching point" of a meta-node ν_i is the first node that has more than one branch if you follow the tree up to the root, starting from ν_i. We will refer to the branching point of a meta-node ν as $bp(\nu)$. Figure A.2 depicts this concept.

If G_c has nodes which do not belong to any cluster, either a meta-node for each node or for a group of nodes is constructed. After $\mathcal{G}$ has been built each node has to belong exactly to one meta-node. Edges between a pair of clusters are collapsed into one meta-edge. It is worth mentioning that the meta-edges do not correspond

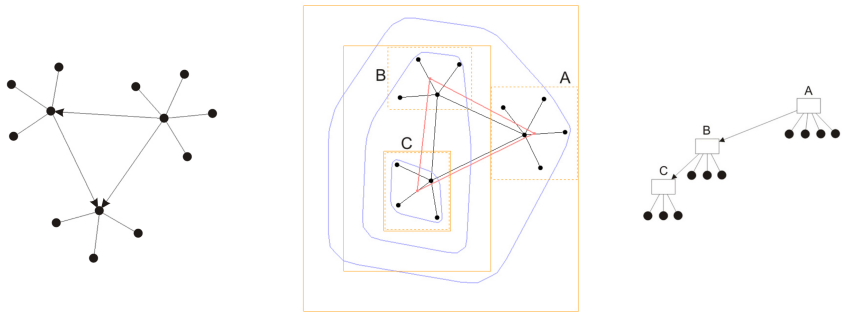


Figure A.3.: A sample graph (left), its corresponding meta-graph (middle) and the respective hierarchy (right). Outer bounding boxes are shown in solid orange and inner bounding boxes dashed. The red lines depict the meta-edges.

to the branches in the tree. Figure A.3 shows a sample graph, its corresponding meta-graph and the tree structure defining the hierarchy of clusters (and meta-nodes respectively).

A.3.2. Meta Layouter

The proposed algorithm follows the approach of Fruchterman and Reingold [7] including their force model, although other force models could be used as well. The spring forces are therefore computed with

$$\begin{aligned} f_a &= \frac{d^2}{k} \\ f_r &= -\frac{k^2}{d} \end{aligned} \tag{A.1}$$

where f_a and f_r are the attractive and repulsive forces, respectively. As with classical FDP algorithms repulsive forces are calculated between each pair of meta-nodes and attractive forces only between meta-nodes which are connected by a meta-edge. d is the current length and k is the so-called dynamic natural length (as Li et al. [12] called it). The dynamic natural length between two meta-nodes ν_1 and ν_2 is

$$k = \frac{l_1}{2} + \frac{l_2}{2} \tag{A.2}$$

where l_1 is the diagonal of either the inner or outer bounding box of ν_1 and l_2 is the diagonal from b_{inner} or b_{outer} of ν_2. The current length d between two meta-nodes is simply the distance of the midpoints of the appropriate bounding boxes. Which bounding box to consider depends on ν_1 and ν_2. Two different cases can be distinguished:

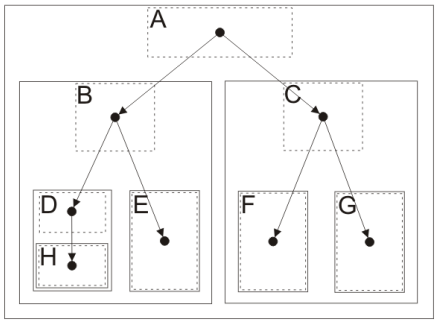


Figure A.4.: An example of a hierarchical clustergraph, showing the bounding boxes as well as the underlying rooted tree.

bp(ν_1) and bp(ν_2) are unequal: In this case we construct a path $(\nu_1, ..., \nu_{c1}, \nu_c)$ from ν_1 to the first node ν_c which has both metanodes, ν_1 and ν_2, in common. Analogous we construct a path $(\nu_2, ..., \nu_{c2}, \nu_c)$ for ν_2. The outer bounding boxes for ν_{c2} and ν_{c1} are then considered. As shown in Figure A.4 this would for example be the case for meta-node D and F, where the outer bounding boxes of B and C have to be taken.

bp(ν_1) equals bp(ν_2): This means that ν_1 and ν_2 are accommodated by the same superordinate meta-node. First, it has to be checked if ν_1 and ν_2 are in different subtrees or in the same subtree of $bp(\nu_1)$. In the former case the outer bounding box of the subordinate meta-nodes of $bp(\nu_1)$ of the subtrees where ν_1 and ν_2 reside must be taken. For instance, in Figure A.4 meta-nodes E and H share the same branching point B but are in different subtrees, therefore the outer bounding boxes of D and E have to be used. If they are in the same subtree it is sufficient to consider the inner bounding box of ν_1 and ν_2.

After all nodes have been displaced at the end of an iteration, the bounding boxes have to be recalculated. Since the outer bounding box of meta-nodes in higher levels depend on the outer bounding box of the meta-nodes below, this can be done recursively, starting from the leaves of the tree upwards. Note that the size of the inner bounding box does not change, because the layout of the nodes of the meta-node is not changed. Therefore only the position of the inner bounding box has to be updated.

A.3.3. Break Condition

The calculation can safely be terminated if none of the bounding boxes intersect, although the resulting layout could still be contracted. Determining if two axis aligned bounding boxes (AABBs) A and B overlap is simple. We only have to check if $A_x^{min} > B_x^{max}$ or $B_x^{min} > A_x^{max}$ or $A_y^{min} > B_y^{max}$ or $B_y^{min} > A_y^{max}$ (see for example [15]) to determine if A and B are disjoint. The only problem which remains is to determine which bounding boxes have to be checked for overlapping. Because

sometimes it is valid that a bounding box is inside another one (for example b_{inner} of E inside b_{outer} of B in Figure A.4), in other cases it would be wrong (for instance b_{inner} of F is not allowed to be inside b_{outer} of B).

Each meta-node ν_i is checked against all metanodes ν_j, $j = 1, .., i-1$. To find out if the inner or outer bounding box has to be used the same cases as in Section A.3.2 are distinguished, except this time we additionally check if $\nu_\mathbf{1}$ is a subcluster of $\nu_\mathbf{2}$ or vice versa. If these particular cases are not handled separately the break condition would also break if the superordinate metanode is inside the outer bounding box of the subordinate node. For instance in Figure A.3 the algorithm would also stop if A is inside b_{outer} of B. It should be noted that these additional cases can also be considered in the meta layouter with the downside of getting slightly larger graph embeddings.

A.3.4. Closing and Opening of Clusters

Clusters can be closed or opened dynamically. This allows the user to hide certain parts of a large graph in which he is not currently interested. If a cluster C_i gets closed, the size of the bounding box and the boundary are scaled down by a certain factor f in regard to the center of the outer bounding box and is marked as *closed*. All subclusters of C_i are marked as *hidden* and their bounding boxes remain unchanged. *Hidden* subclusters are completely suppressed, neither their outline nor their corresponding nodes and edges containing these nodes are shown. For a *closed* cluster only its content is invisible.

Since we can be sure that the layout is non-overlapping (meta-nodes corresponding to *hidden* clusters are excluded) only the attractive part of the meta spring embedder is executed to contract the layout. This keeps changes local, preserving the mental map ([14]). If due to this process a closed cluster is moved, all of its hidden subclusters are moved accordingly (this guarantees that if a *closed* cluster is opened again all of its subclusters are contained in its outer bounding box).

If a cluster gets opened, its bounding box is scaled by the factor $1/f$ and is marked *visible* as well as its subclusters. However, the situation can also be handled differently. For instance, subclusters may only get tagged visible until a *closed* cluster is encountered. This allows successive reopening of *closed* clusters. This time both the attractive part and the repulsive part are executed (since there can be overlapping bounding boxes once a cluster is opened). Figure A.5 illustrates this concept where a cluster is closed, moved during the layout process and opened again (see also Figure A.7).

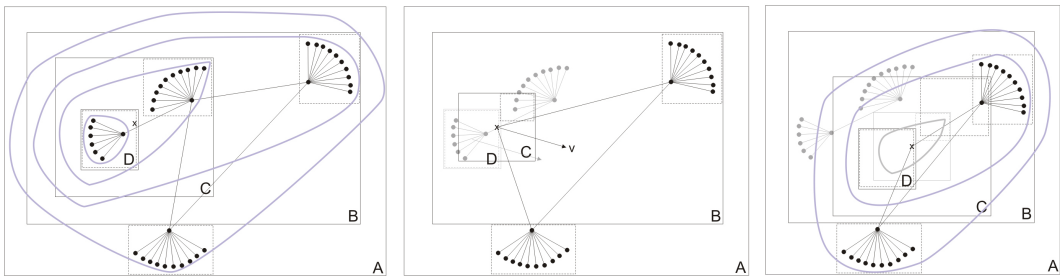


Figure A.5.: Left: An initial layout of a graph with four clusters A, B, C and D. Middle: Cluster C is closed by 50%, therefore both bounding boxes of C are scaled in regard to the center of its outer bounding box. The bounding boxes of the subcluster D remain unaltered. D and its contents as well as the nodes of C are just hidden (depicted grey in the figure). The force directed placement algorithm displaces C by the vector v, therefore the same displacement is carried out on D. Right: C at its final position after the FDP algorithm has finished. When C gets openend the bounding boxes are rescaled to the original size. The subcluster's bounding boxes are automatically at the right position. However, the inflated bounding box may overlap other (C overlaps b_{inner} of B).

A.3.5. Automatic Clustering

We implemented an algorithm for automatic detection of clusters in a directed graph which showed to be useful in regard to semantic nets. In our case each cluster C_i has a unique node designated as cluster center n_c^i.

In a first step, this cluster centers are identified by finding nodes which fulfill the condition $o(n) - i(n) > t$ where t is a user-defined threshold and $i(n)$ is the in-degree of a node n, defined as the number of directed edges pointing to n, mathematically

$$i(n) = |\{m|(m,n) \in E\}| \tag{A.3}$$

Similarly, the out-degree $o(n)$ is defined as

$$o(n) = |\{m|(n,m) \in E\}| \tag{A.4}$$

Furthermore, nodes which lie within a graph-theoretical distance[1] d – also a user specified value – from a cluster center n_c^i are assigned to the node set N_C^i of the corresponding cluster C_i. Note that for the time being a node can belong to different clusters.

Second, if a cluster center n_c^i is part of the set of nodes of a cluster C_k, the cluster C_i may be a subcluster of C_k and is therefore marked as possible candidate. We will refer to the possible subcluster candidates of a cluster C_i as $pc(C_i)$. The union set

[1] the length of the shortest path between two vertices in a graph

$N_C^i \cup N_C^k$ is removed from N_C^k to ensure that those nodes only belong to exactly one cluster.

Next, the subcluster hierarchy is built from the possible candidates determined in Step 2. For example, in Figure A.3 cluster C is a possible candidate for a subcluster of A as well as B. However, only one predecessor is allowed for C. To determine the correct predecessor of a cluster C_i we have to find a cluster C_k where $C_i \in pc(C_k)$ but $\nexists C_1 \in pc(C_k)$, $C_1 \neq C_i$ with $C_i \in pc(C_1)$ and $\nexists C_2 \in pc(C_1)$, $C_2 \neq C_i$ with $C_i \in pc(C_2)$ and so on. For the given example C must be a subcluster of B because $pc(A) = \{B, C\}$ contains the cluster B which also has C as possible candidate.

Finally, nodes n are resolved which still belong to more than one cluster

$$C_s = \{C_1, C_2, ..., C_n\} \tag{A.5}$$

In such a case we base our decision on the number of edges

$$d(n, C_i) = |\{m|(n, m) \in E \vee (m, n) \in E, m \in C_i\}| \qquad \text{with } C_i \in C_s \tag{A.6}$$

which connect such a node n with other nodes of the clusters in question C_s. If n has the same number of connections with nodes of different clusters, mathematically $d(n, C_i) = k \; \forall i = 1, .., n$ we remove the node from all clusters in C_s and assign it to a possibly existing superordinate cluster. Otherwise a cluster $C_i \in C_s$ must exist, for which $d(n, C_i)$ is maximal. In this case n is assigned to C_i and removed from all clusters in the set $C_s \setminus C_i$.

A.4. Applications

The presented method was written in C++ and is successfully used in the Melvil® Knowledge Exploration Software by uma information technology GmbH. Melvil is a tool for the creation of semantic networks in terms of directed graphs, concepts and clusters. The visualization of such knowledge networks (or semantic networks) for knowledge exploration highly depends on methods to display the elements such as nodes, clusters and relations in a clear manner. The arrangement of these clusters is implemented via the aforementioned techniques. Exploration of networks is also supported in Melvil, via search queries, individualized visualization of clusters and by hiding certain parts of the graph (e.g. closing of clusters).

However, since the presented technique is a layout adjustment method, the quality of a final embedding depends on the initial layout. Figure A.6 through Figure A.8 show results achieved with the proposed algorithm. Although the repulsive part

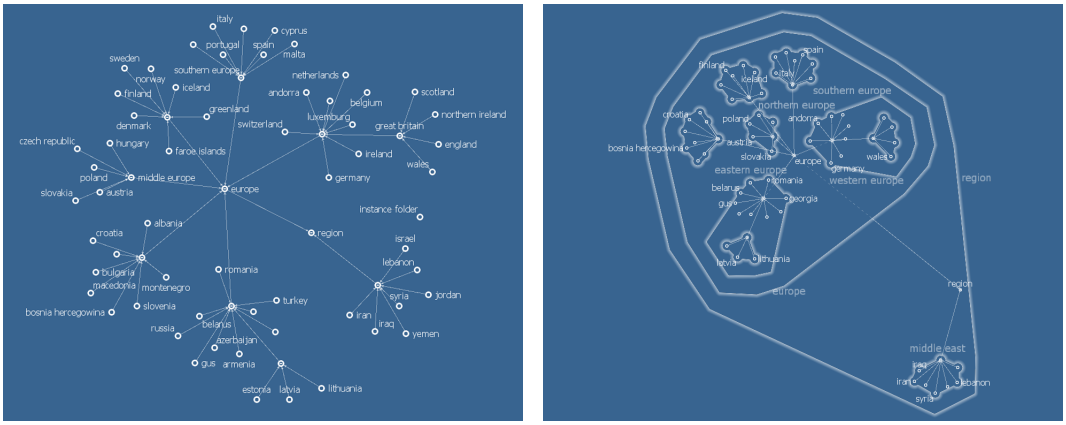


Figure A.6.: A graph representing various countries of Europe and Middle East. An embedding of the graph without clustering is shown on the left. Annotating this graph with clustering information for certain regions (Western Europe, Eastern Europe, ...) leads to a visually more comprehensible representation (right). The cluster centers were detected automatically.

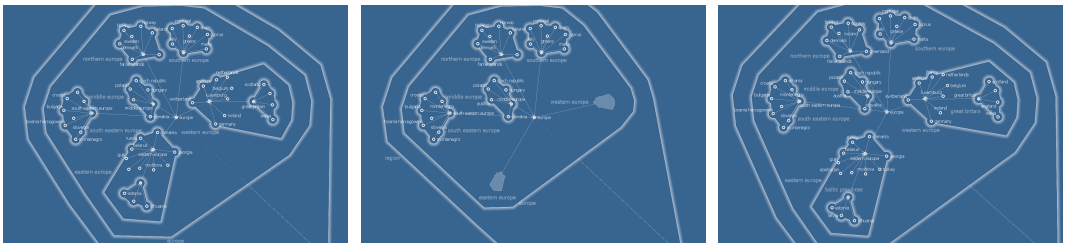


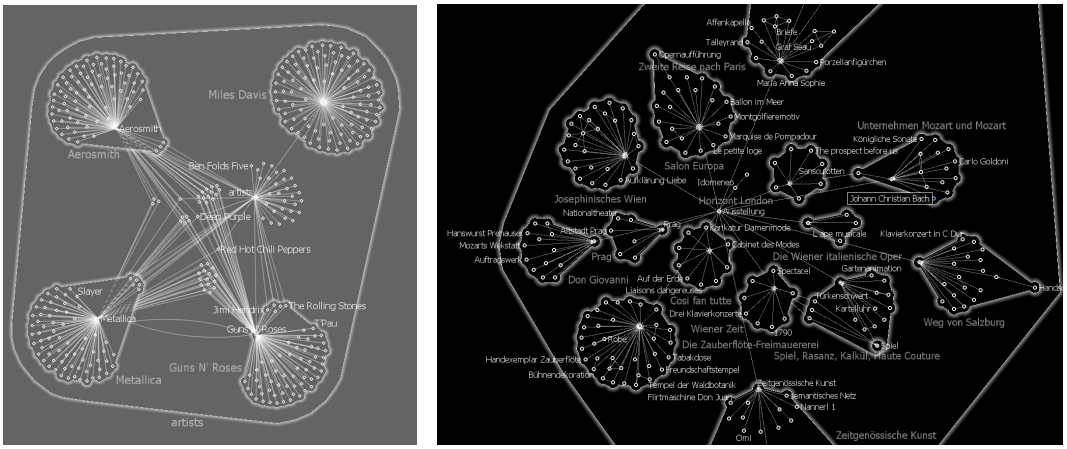
Figure A.7.: The initial layout of the graph is shown on the left. The clusters *Western Europe* and *Eastern Europe* are closed (middle). Their boundary are shrunken to 20% of their original size and their contents are hidden. If the clusters are reopend again the inflated clusters are arranged properly (right).

of the algorithm has runtime of $O(|C|^2)$, the algorithm performed well in practice, because usually $|C| \ll |V|$.

A.5. Future Work

The main problem is the break condition of the algorithm. In some cases it aborts the calculation too early, yielding a layout with long distances between meta-nodes. We currently experiment with separating the repulsive and attractive forces. That means, in a first step only repulsive forces are working until a layout with non-overlapping meta-nodes is reached. Second, the meta-nodes are contracted by step-wise reducing the edge length by a certain amount as long as no overlapping occurs.

Another problem lies in the calculation of the dynamic natural length, which restricts



(a) A graph showing the interconnections between various music artists.

(b) Close-up view of a graph describing the life and work of Wolfgang Amadeus Mozart.

Figure A.8.: Two example graphs showing results achieved with our algorithm.

the closeness of two meta-nodes. In many cases this distance is too far. A solution could be to determine the length more precisely by measuring the distance from the midpoint to the AABB. K-dops [11] would approximate the convex hull of the node set more precisely than AABBs leading to a better estimation of the natural length. The overlap test between two k-dops is a general version of the AABB[2] collision check (see Section A.3.3). In the overlapping case maximal $k/2$ interval tests are necessary. Furthermore the optimal bounding volume for a cluster in the hierarchy can be easily computed by merging the bounding volumes of the subordinate clusters. Therefore a more compact layout with negligible computational overhead would be possible.

As shown by Frishman and Tal [5] the high computational power of GPUs can be harnessed to accelerate force directed layout algorithms. We are planning to make a GPU implementation of the presented meta layouter based on the work of Frishman and Tal [5, 6] which also performs the collision detection on graphics hardware.

By now, only the force model by Fruchterman and Reingold [7] has been used. It would be of interest to compare the results with results reached with other force models, for example the one proposed by Eades et al. [4] or Creek [2].

A.6. Conclusions

We have presented a force-directed layout adjustment algorithm for the drawing of hierarchies of clusters. Cluster hierarchies allow the user to adopt the granularity of the graph to their specific needs. Users can therefore suppress parts of the graph which are currently not of concern. Calculating an overlapping free layout with FDP

[2] In fact, an AABB in two dimensions is a special case of a k-dop with $k = 4$

requires some attention when determining the ideal length of edges. Two bounding boxes ensure that a cluster is not placed inside the boundary of another cluster nor in a cluster hierarchy where it does not belong. Before the layout adjustment can start, a meta graph is derived based on the initial layout and clustering information. The main motivation for this work is from the domain of semantic net exploration and knowledge space visualisation. Examples from these fields were shown.

A.7. Acknowledgments

The context and idea of this work has its origins in a research project by uma information technology GmbH. The author would like to thank uma information technology GmbH for providing the image for Figure A.8(b) and the graphs used in Figure A.6, A.7 and A.8(a).

A.8. Bibliography

[1] Brockenauer, R., Cornelsen, S.: *Drawing clusters and hierarchies.* In: M. Kaufmann, D. Wagner (eds.) Drawing Graphs: Methods and Models, 2025, pp. 193–227. Springer-Verlag, Berlin, Germany (2001)

[2] Creek, A.: *Forces of nature* (2001). Available online: `http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2001/hons_0102.pdf`

[3] Eades, P.: *A heuristic for graph drawing.* Congressus Nutnerantiunt **42**, 149–160 (1984)

[4] Eades, P., Feng, Q.W., Lin, X.: *Straight-line drawing algorithms for hierarchical graphs and clustered graphs.* Proceedings of the Symposium on Graph Drawing pp. 113–128 (1996)

[5] Frishman, Y., Tal, A.: *Dynamic drawing of clustered graphs.* EuroVis pp. 75–82 (2007)

[6] Frishman, Y., Tal, A.: *Multi-level graph layout on the gpu.* Proceedings Information Visualization (2007)

[7] Fruchterman, T.M.J., Reingold, E.M.: *Graph drawing by force-directed placement.* Software - Practice and Experience **21**, 1129–1164 (1991). Available online: `http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol21/issue11/spe060tf.pdf`

[8] Fung, G.: *A comprehensive overview of basic clustering algorithms* (2001). Available online: `www.cs.wisc.edu/~gfung/clustering.pdf`

[9] Huang, M.L., Eades, P.: *A fully interactive system for clustering and navigating large graphs.* Graph Drawing, Springer Lecture Notes in Computer Science pp. 374–383 (1998)

[10] Kamada, T., Kawai, S.: *An algorithm for drawing general undirected graphs.* Information Processing Letters **31**, 7–15 (1989)

[11] Klosowski, J.T., Held, M., Mitchell, J.S.B., Sowizral, H., Zikan, K.: *Efficient collision detection using bounding volume hierarchies of k-DOPs.* IEEE Transactions on Visualization and Computer Graphics **4**(1), 21–36 (1998). Citeseer.ist.psu.edu/klosowski96efficient.html

[12] Li, W., Eades, P., Nikolov, N.: *Using spring algorithms to remove node overlapping* (2006). Available online: `http://crpit.com/confpapers/CRPITV45Li.pdf`

[13] Macqueen, J.B.: *Some methods of classification and analysis of multivariate observations.* In: Proceedings of the Fifth Berkeley Symposium on Mathemtical Statistics and Probability, pp. 281–297 (1967)

[14] Misue, K., Eades, P., Lai, W., Sugiyama, K.: *Layout adjustment and the mental map.* Journal of Visual Language and Computing **6** (1995)

[15] Möller, T., Haines, E.: *Real-Time Rendering*, first edn. A K Peters (1999)

[16] Nagamochi, H., Kuroya, K.: *Convex drawing for c-planar biconnected clustered graphs.* In: G. Liotta (ed.) Graph Drawing, Perugia, 2003, pp. pp. 369–380. Springer (2004)

[17] do Nascimento, H.A.D., Eades, P.: *A system for graph clustering based on user hints.* Selected papers from the Pan-Sydney workshop on Visualisation **2**, 73–74 (2000). Available online: `http://crpit.com/confpapers/CRPITV2Nascimento.pdf`

[18] Nizar, G., Michel, C., Nozha, B.: Unsupervised and semi-supervised clustering: a brief survey (2005). Citeseer.ist.psu.edu/727015.html

[19] Quinn, N., Breuer, M.: *A force directed component placement procedure for printed circuit boards.* IEEE Transactions on Circuits and Systems pp. 377–388 (1979)

[20] Wagstaff, K., Cardie, C., Rogers, S., Schrödl, S.: *Constrained k-means clustering with background knowledge.* In: ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning, pp. 577–584. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

[21] Wang, X., Miyamoto, I.: *Generating customized layouts.* Proceedings of the Symposium on Graph Drawing (GD 1995) pp. 504–515 (1995)