ISSN 1213-6972 *Volume 21, Number 2, 2013*

Journal of WSCG

An international journal of algorithms, data structures and techniques for computer graphics and visualization, surface meshing and modeling, global illumination, computer vision, image processing and pattern recognition, computational geometry, visual human interaction and virtual reality, animation, multimedia systems and applications in parallel, distributed and mobile environment.

Vaclav Skala – Union Agency

Journal of WSCG

Editor-in-Chief: Vaclav Skala
c/o University of West Bohemia
Faculty of Applied Sciences
Univerzitni 8
CZ 306 14 Plzen
Czech Republic
http://www.VaclavSkala.eu http://www.wscg.eu

Managing Editor: Vaclav Skala

Printed and Published by:
Vaclav Skala - Union Agency
Na Mazinach 9
CZ 322 00 Plzen
Czech Republic

Hardcopy: ***ISSN 1213 – 6972***
CD ROM: ***ISSN 1213 – 6980***
On-line: ***ISSN 1213 – 6964***

Journal of WSCG

Editor-in-Chief

Vaclav Skala

c/o University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering
Univerzitni 8
CZ 306 14 Plzen
Czech Republic

http://www.VaclavSkala.eu
Journal of WSCG URLs: http://www.wscg.eu or http://wscg.zcu.cz/jwscg

Editorial Advisory Board
MEMBERS

WSCG 2013

Board of Reviewers

Agathos, Alexander
Assarsson, Ulf
Ayala, Dolors
Backfrieder, Werner
Barbosa, Joao
Barthe, Loic
Battiato, Sebastiano
Benes, Bedrich
Benger, Werner
Bilbao, Javier,J.
Biri, Venceslas
Birra, Fernando
Bittner, Jiri
Bosch, Carles
Bourdin, Jean-Jacques
Brun, Anders
Bruni, Vittoria
Buehler, Katja
Bulo, Samuel Rota
Cakmak, Hueseyin
Camahort, Emilio
Casciola, Giulio
Cline, David
Coquillart, Sabine
Cosker, Darren
Daniel, Marc
Daniels, Karen
de Geus, Klaus
de Oliveira Neto, Manuel Menezes
Debelov, Victor
Drechsler, Klaus
Durikovic, Roman
Eisemann, Martin
Erbacher, Robert
Feito, Francisco
Ferguson, Stuart
Fernandes, Antonio
Fuenfzig, Christoph
Gain, James
Galo, Mauricio
Gobron, Stephane
Grau, Sergi
Gudukbay, Ugur
Guthe, Michael
Hansford, Dianne
Haro, Antonio
Hasler, Nils
Hast, Anders
Hernandez, Benjamin
Hernandez, Ruben Jesus Garcia
Herout, Adam
Herrera, Tomas Lay
Hicks, Yulia
Hildenbrand, Dietmar
Hinkenjann, Andre
Chaine, Raphaelle
Choi, Sunghee
Chover, Miguel
Chrysanthou, Yiorgos
Chuang, Yung-Yu
Iglesias, Jose,A.
Ihrke, Ivo
Iwasaki, Kei
Jato, Oliver
Jeschke, Stefan
Jones, Mark
Juan, M.-Carmen
Kämpe, Viktor
Kanai, Takashi
Kellomaki, Timo
Kim, H.
Klosowski, James
Kolcun, Alexej
Krivanek, Jaroslav
Kurillo, Gregorij
Kurt, Murat
Kyratzi, Sofia
Larboulette, Caroline
Lee, Jong Kwan Jake
Liu, Damon Shing-Min
Lopes, Adriano
Loscos, Celine
Lutteroth, Christof
Maciel, Anderson
Mandl, Thomas
Manzke, Michael
Marras, Stefano
Masia, Belen
Masood, Syed Zain
Max, Nelson
Melendez, Francho
Meng, Weiliang
Mestre, Daniel,R.
Metodiev, Nikolay Metodiev
Meyer, Alexandre
Molina Masso, Jose Pascual
Molla, Ramon
Montrucchio, Bartolomeo
Morigi, Serena
Muller, Heinrich
Munoz, Adolfo
Murtagh, Fionn
Okabe, Makoto
Oyarzun, Cristina Laura
Pan, Rongjiang
Papaioannou, Georgios
Paquette, Eric
Pasko, Galina
Patane, Giuseppe
Patow, Gustavo
Pedrini, Helio
Pereira, Joao Madeiras
Peters, Jorg

Pina, Jose Luis
Platis, Nikos
Post, Frits,H.
Puig, Anna
Rafferty, Karen
Renaud, Christophe
Reshetouski, Ilya
Reshetov, Alexander
Ribardiere, Mickael
Ribeiro, Roberto
Richardson, John
Rojas-Sola, Jose Ignacio
Rokita, Przemyslaw
Rudomin, Isaac
Sacco, Marco
Salvetti, Ovidio
Sanna, Andrea
Santos, Luis Paulo
Sapidis, Nickolas,S.
Savchenko, Vladimir
Seipel, Stefan
Sellent, Anita
Shesh, Amit
Sik-Lanyi, Cecilia
Sintorn, Erik
Skala, Vaclav
Slavik, Pavel
Sochor, Jiri
Sourin, Alexei
Sousa, A.Augusto
Sramek, Milos
Stroud, Ian
Subsol, Gerard
Sundstedt, Veronica
Szecsi, Laszlo
Teschner, Matthias
Theussl, Thomas
Tian, Feng
Tokuta, Alade
Torrens, Francisco
Trapp, Matthias
Tytkowski, Krzysztof
Umlauf, Georg
Vasa, Libor
Vergeest, Joris
Vitulano, Domenico
Vosinakis, Spyros
Walczak, Krzysztof
WAN, Liang
Wu, Shin-Ting
Wuensche, Burkhard,C.
Wuethrich, Charles
Xin, Shi-Qing
Xu, Dongrong
Yoshizawa, Shin
Yue, Yonghao
Zalik, Borut
Zara, Jiri
Zemcik, Pavel
Zhang, Xinyu
Zhao, Qiang
Zheng, Youyi
Zitova, Barbara
Zwettler, Gerald

Journal of WSCG

Vol.21, No.2

Contents

	Pages
Suarez,J., Belhadj,F., Boyer,V.: GPU Real Time Hatching	97
Ramos,J., Larboulette,C.: A Muscle Model for Enhanced Character Skinning	107
Kellomäki,T.: Interaction with Dynamic Large Bodies in Efficient, Real-Time Water Simulation	117
Nikodym,T., Havran,V., Bittner,J.: Multiple Live Video Environment Map Sampling	127
Akagi,Y., Furukawa,R., Sagawa,R., Ogawara,K., Kawasaki,H.: Marker-less Facial Motion Capture based on the Parts Recognition	137
Lobachev,O., Schmidt,M., Guthe,M.: Optimizing Multiple Camera Positions for the Deflectometric Measurement of Multiple Varying Targets	145
Günther,C., Kanzok,Th., Linsen,L., Rosenthal,P.: A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds	153

GPU real time hatching

Suarez Jordane
Université Paris 8
2 rue de la liberté
93526, Saint Denis,
France
suarez@ai.univ-paris8.fr

Belhadj Farès
Université Paris 8
2 rue de la liberté
93526, Saint Denis,
France
amsi@ai.univ-paris8.fr

Boyer Vincent
Université Paris 8
2 rue de la liberté
93526, Saint Denis,
France
boyer@ai.univ-paris8.fr

ABSTRACT

Hatching is a shading technique in which tone is represented by a series of strokes or lines. Drawing using this technique should follow three criteria: the lighting, the object geometry and its material. These criteria respectively provide tone, geometric motif orientation and geometric motif style. We present a GPU real time approach of hatching strokes over arbitrary surfaces. Our method is based on a coherent and consistent model texture mapping and takes into account these three criteria. The triangle adjacency primitive is used to provide a coherent stylization over the model. Our model computes hatching parameter per fragment according to the light direction and the geometry and generates hatching rendering taking into account these parameters and a lighting model. Dedicated textures can easily be created off-line to depict material properties for any kind of object. As our GPU model is designed to deal with texture resolutions, consistent mapping and geometry in the object space, it provides real time rendering while avoiding popping and shower-door effects.

Keywords

non-photorealistic rendering, hatching, stroke based texture, shading, GPU

1 INTRODUCTION

Hatching is an artistic drawing technique that consists in drawing closely spaced lines to represent objects. Artists depict tonal or shading effects with short lines. Parallel lines or cross lines can be used to produce respectively hatching or cross-hatching. Hatching is commonly used in artistic drawing such as comics but is also largely used in specific visualization systems such as archeology, industry and anatomy. Hatching can also be used as a common way to promote any kind of products. Hatching generally refers only to lines or strokes. We prefer the term of geometric motifs including various possible patterns and dots, lines or cross lines. Artists can thus provide hatching, cross-hatching as well as stippling.

Hatching is produced by the artists following three criteria: lighting, object geometry and object material. These criteria provide tone, geometric motif orientation and geometric motif style. The tone of the geometric motif used for hatching refers to the lighting equation. Geometric motif orientation is provided by the object geometry and the direction of light. Finally, depending on the object material, different geometric motifs can be used.

This paper presents a method for real time hatching considering these three criteria. Our model is relevant in all interactive applications such as games, scientific, educational or technical illustrations. As it is often mentioned in non-photorealistic rendering (NPR) work, the lack of temporal coherence in stroke placements and image-based approaches respectively produce flickering and shower-door effect during animations.

We deal with objec-space coherence by considering that stroke width and density should be adjusted to camera, lighting and scene transformation. Desired tones should be maintained even as the object moves toward or away from the camera. We propose a full GPU implementation able to generate hatching strokes on 3D scene. As [Pra01], we address the same challenges: (1) limited run-time computations, (2) frame-to-frame coherence among strokes, and (3) control over stroke size and density under dynamic viewing conditions.

But by opposition to the previous work, we consider that the stroke orientations directly depend on the light direction and should be computed per fragment. In

fact, as mentioned above, artists choose the geometric motif orientation according to the geometry and the light direction. As artists do, we provide a model able to consider these two constraints. Rare are the previous work in which the light direction is considered. In our approach, we address these challenges by providing a new texture mapping model that deals with modern geometry stage hardware.

In this paper, we provide a hatching rendering that satisfies the three criteria. This is achieved noticeably through:

- Hatching parameter computations per fragment according to the light direction and the triangles topology;
- Generation of hatching renderings that take into account these parameters and a lighting model.

We first present the related work in NPR domain dedicated to stroke based renderings. Then we present our model and its implementation. The performance of our model are discussed and the results demonstrate the relevance of our solution in the hatching problem. Finally, we conclude and propose future work.

2 PREVIOUS WORK

Hatching can be considered as a subset of drawing or painting style in which individual strokes are produced. Most related work have been presented as NPR papers. For that reason, we present hereafter related work on stroke based NPR.
Stroke based NPR can be categorized into two kinds of methods depending on the type of treated primitives. Some focus on pixel buffers while others deal with geometric objects. Below, we present general stroke based methods according to this classification, image space or object space, and particularly focus on specific hatching ones.

Image space methods use images in order to extract color, depth or tone information. In this kind of approaches the main challenge lies in geometric motif orientation:
In [Hae90], P. Haeberli has presented a method that creates impressionist paintings from images using "brush-strokes". The orientation of strokes depends on image gradient and direction fields.
In [Sal94], the proposed method aims to generate pen and ink illustrations from images by placing stroke textures. The orientation of these textures is computed according to the image gradient and the user defined directions.
In [Win96], the authors have created pen and ink rendering from a viewpoint of 3D scenes. To achieve this goal, the method uses controlled density hatching and user defined directions.
The model proposed in [Sal97] produces pen and ink style lines from images or a viewpoint of 3D models. In this model, strokes are oriented according to the direction fields.
In [Sai90], the authors have proposed to produce line drawing illustration from a viewpoint of 3D scenes according to curved hatchings and "G-Buffer". This specific geometric buffer contains geometric surface properties such as the depth or the normal vector of each pixel. The curvature is extracted for each object and indicates its topology. Hatching is then automatically produced using curvature and other information like rotations or texture coordinates.

The main drawback of these methods remains the appearance of the shower-door effect. In fact, as the geometry of these scenes is rarely considered, lighting, depth and other 3D information are almost never taken into account.
Then, we prefer approaches that focus on 3D model information.

In object space methods, additional information such as light position, depth and normal of each vertex can be extracted from geometry:
In [Deu00], the proposed method generates pen and ink illustrations using different styles such as the cross hatching and according to the object normals and the scene viewpoint.
The method presented in [Kap00] creates some artistic renderings particularly "geograftals" strokes (oriented geometric textures) according to the principal curvature of the 3D models.
In [Lak00], Lake & al. have presented a method that produces stylized renderings like pencil and stroke sketch shading which orientation depends method also allows the user to create cartoon rendering of 3D models.
In [Her00], the authors have proposed a method to generate line art renderings using hatch mark and generated smooth direction fields.
In [Pra01], non-photorealistic renderings with textures of hatching strokes called "tonal art map"(TAM) are described. Multitexturing is used for rendering TAMs with spatial and temporal coherence. Orientation of textures is computed according to a curvature based on direction fields and lapped texture parametrization (set of parametrized patches).
In [Web02], Webb & al. have presented two real time hatching schemes based on TAM. This method avoids blending or aliasing artifacts using a finer TAM resolution. We can notice that one of this scheme supports colored hatching.

In [Coc06], a model for pen and ink illustrations using real time hatching shadings is proposed. This approach is hybrid and combines image space and object space stroke orientation methods. Stroke orientations are computed according to a combination of dynamic object space parameterizations and view dependent ones. It permits a better spatial coherence than the object-space approaches and reduces the shower-door effect compared to image-based hatching methods but is dedicated to specific geometries such as tree models.

The main drawback of this kind of methods is the texture discontinuity. As texture coordinates are independently generated for each triangle, a possible discontinuity may be introduced and visible artifacts may be generated.

3 OUR MODEL

We aim to produce hatchings according to the previously described three criteria. Tone, form and material should provide a coherent and consistent hatching. As described in [Pra01], we think that a texture based approach is suitable to achieve the hatching and different materials can be depicted by the texture variety. Form and tone should be the result of a lighting model but by opposition to the related work, in our approach, the geometric motif (i.e strokes or lines for example) orientation also depends on the light direction and not only on the model curvature. As a consequence, textures should contain set of continuous tones.
We detail our solution realized in four steps:

1. The generation of textures representing tones: each generated texture represents a tone composed by a geometric motif (for example a set of lines, strokes or points). The set of generated textures should contain a set of continuous tones (see figure 1). Note that to compute a tone per fragment, we should determine a texture number and texture coordinates.

2. Texture orientation and texture coordinates generation: according to the light position and for each triangle, textures should be oriented in order to follow the light displacement. Texture coordinates generation must guarantee no texture distortion for the three vertices of each triangle. At this step, texture coordinates are computed for a given primitive and for each vertex (see section 3.2 and figure 3). This provides the first part of the paper contribution.

3. Texture continuity: we ensure the texture continuity between each triangle and its neighbors by computing a blend factor per neighbor (see section 3.3 and figure 8). This step permits a full computation taking into account each triangle and its neighbors and constitutes the second part of this paper contribution.

4. Tonal mapping: we compute the tone per fragment by interpolating vertex texture coordinates (step 2) and a light equation determining the texture number. This last computation is realized per fragment since the texture number is not obtained by interpolation (see figure 9). Finally, for a triangle, we blend results provided by neighbors according to the blend factor computed at step 3. Note that for a given fragment in a triangle, blend factors are interpolated (see figure 8).
Moreover, even if we present our model with a set of textures, in practice, we have a multi-resolution set of textures providing mipmapping and avoiding aliasing (see figure 2). This last step generates the final rendering using previously computed values, a lighting model and a multi-resolution tonal art map.

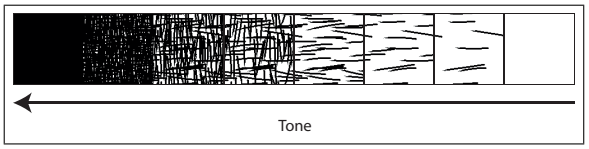


Figure 1: Tonal art map example composed by 8 different textures.

3.1 Texture generation

In our method, we use precomputed tonal art map introduced in [Pra01], where hatch strokes are mipmapped images corresponding to different tones.
Tonal art map must satisfy some consistency constraints:

- For a given tone, all textures must have the same grayscale average;
- geometric motifs present in a given texture must be present in all higher texture resolutions and superior tones;
- geometric motifs have the same form regardless to the texture resolution. For example, lines must have the same width and length and points must have the same radius.

As an example, figure 1 shows a set of continuous tones given at a high resolution (TAM). Figure 2 presents an example of multi-resolution tonal art map: a multi-resolution tone texture deduced from figure 1. Note that, as all mipmapping techniques, the multi-resolution TAM will be used to avoid aliasing by taking into account the fragment depth. So, as it is a well-known solution, we do not consider this problem hereafter and use only figure 1 throughout the article to illustrate our purpose. Once we have this texture palette, we want to determine the texture coordinates of each vertex of our 3D model.

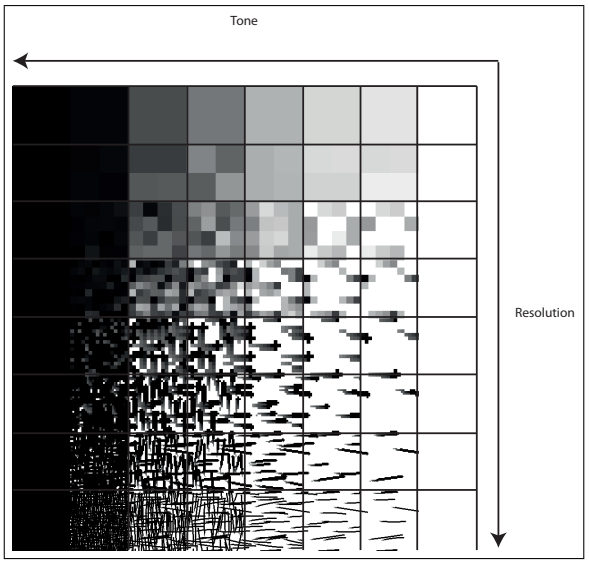


Figure 2: Multi-resolution tonal art map example.

3.2 Hatching texture orientation

As described above, our approach matches the geometric motif orientation depending on the light direction. Thus, as shown on figure 3 considering a given light source and a 3D model, we have to orient and crop the TAM according to the light source properties (position, direction, cut-off and spot exponent).

To describe our model, we use the following nomenclature. L describes a light, $\overrightarrow{L_d}$ depicts its direction while L_p depicts its position. Note that in the case of directional light, L_p is not used. T_c is a triangle of the 3D model and V_i ($i \in \{0,1,2\}$) are its vertices. Each vertex V_i, represented by its 3D space coordinates (x,y,z), is associated to a hatching parameter corresponding to its 2D texture coordinates noted $H_i^c(s,t)$. Figure 5 illustrates this notation used hereafter in the paper. As mentioned in the introduction of our model description, in this section we present a hatching parameter computation considering only one triangle (*i.e* generalization considering triangle and neighbors is detailed in the next section).

Since we want to guarantee no texture distortion in the considered triangle, the applied transformations should necessary be realized per primitive. This is done by computing, for each triangle, its hatching parameters according to the tangent space of the triangle itself. This tangent space is the Tangent-Binormal-Normal $(\overrightarrow{T},\overrightarrow{B},\overrightarrow{N})$ coordinates system with $\overrightarrow{B}$ representing the vector from the triangle toward the light, projected on the triangle plane. In the case of positional light with its position (L_p) and direction ($\overrightarrow{L_d}$), we compute the direction to the light per triangle ($\overrightarrow{L}$) considering G_c the triangle barycenter (*i.e* $\overrightarrow{L} = \overrightarrow{L_p - G_c}$). In the case of directional light, the direction to the light ($\overrightarrow{L}$) is equal to the opposite of the light direction ($\overrightarrow{L_d}$) (*i.e* $\overrightarrow{L} = -\overrightarrow{L_d}$). Finally, $\overrightarrow{B}$ is the normalized projected vector $\overrightarrow{L}$ in the triangle plane.

Then, for each vertex V_i of each triangle T_c, by applying a change of basis from the object local space to $(\overrightarrow{T},\overrightarrow{B},\overrightarrow{N})$ we obtain, for the corresponding $\overrightarrow{T}$ and $\overrightarrow{B}$ coordinates, the vertex 2D texture coordinates stored in H_i^c as shown in algorithm. 1:

$\overrightarrow{N} \leftarrow$ normalize $((\overrightarrow{V_1 - V_0}) \times (\overrightarrow{V_2 - V_0}))$
if *the light source is positional* **then**
  $G \leftarrow (V_0 + V_1 + V_2)/3$;
  $\overrightarrow{L} \leftarrow \overrightarrow{L_p - G}$;
else
  $\overrightarrow{L} \leftarrow -\overrightarrow{L_d}$;
end
$\overrightarrow{B} \leftarrow$ normalize$(\overrightarrow{L} - \overrightarrow{N} \cdot (\overrightarrow{L} \cdot \overrightarrow{N}))$;
$\overrightarrow{T} \leftarrow \overrightarrow{B} \times \overrightarrow{N}$;
foreach *i in* $\{0,1,2\}$ **do**
  $H_i^c.s \leftarrow V_i \cdot \overrightarrow{T}$;
  $H_i^c.t \leftarrow V_i \cdot \overrightarrow{B}$;
end

Algorithm 1: Computing hatching texture orientation per triangle.

This step is performed in the geometry shader stage, for each triangle T_c expressed in the local object space. Thus, we compute and emit for each V_i its position and texture coordinates.

We obtain, as shown in figure 4-(a), hatchings that take into account triangle orientations. The four triangles presented on this figure have different normals. Remark that hatchings on the common edges between triangles produce stroke discontinuities. These discontinuities are naturally due to the difference of orientation ($\overrightarrow{T}$) computed at this level but also to the difference of shading (which determines tones of the art map) computed at the last level (fragment shader stage). To avoid this discontinuity we propose to use the triangle adjacency information to compute multi-texturing coordinates and blend the result as shown on figure 4-(b). This process is explained hereafter.

3.3 Adjacency blending

In order to ensure a continuity of textures between two neighboring triangles we determine the contribution of adjacent triangles in the rendering of T_c.

As shown in figure 5, triangle adjacency is a geometric primitive composed by six vertices V_i ($i \in \{0,1,2,\alpha,\beta,\gamma\}$) describing four triangles where T_c is the current processed triangle and T_α, T_β and T_γ its adjacents. This primitive is accessible in the geometry shader stage where data of adjacent triangles (as vertex position) are accessible during T_c processing but are not emitted during this same process.

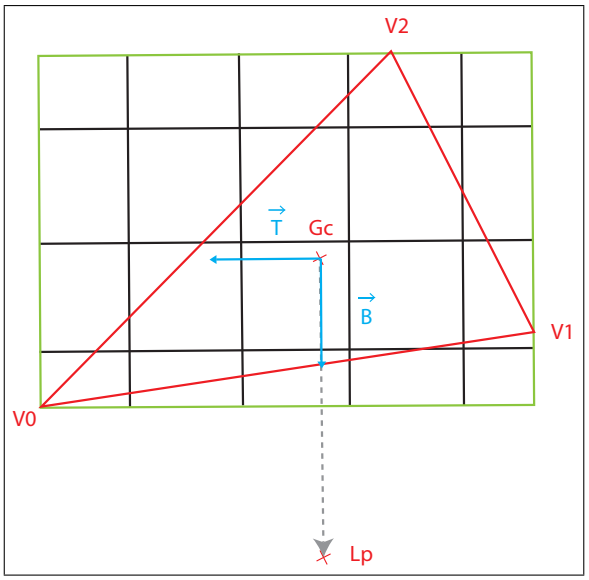
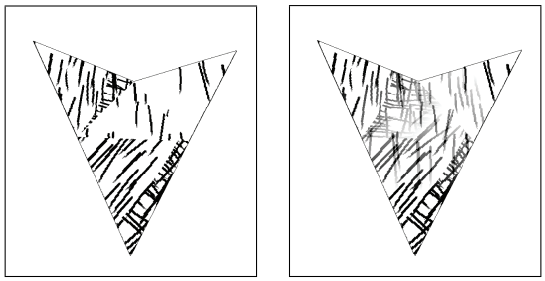


Figure 3: Texture orientation on a given triangle according to the projection of the light position.



(a) (b)

Figure 4: Hatching results on 4 adjacent triangles with different normal vectors. (a): without blend, (b): with blend.

We keep the main calculations of subsection 3.2 by integrating data adjacency, calculating four different texture coordinates per vertex of T_c. In fact, for each vertex V_i ($i \in \{0,1,2\}$) we need to compute the hatching parameters corresponding to each triangle of the adjacency primitive. Thus, we obtain a 3D component H_i^t with $t \in \{c,\alpha,\beta,\gamma\}$ composed by 2D texture coordinates (s,t) and a blend factor f. H_i^t depicts, for the vertex V_i, the contribution of the triangle t in the current triangle hatching. Thus, the triangle is textured according to the orientation of its neighbors to the light. These coordinates are used to mix the results by blending. For each adjacent triangle, we compute its corresponding $(\overrightarrow{T},\overrightarrow{B},\overrightarrow{N})$ coordinates system which we use to compute the corresponding hatching parameters.

Figure 6 illustrates the calculation of hatching parameters where T_c is a triangle being processed and T_γ an adjacent triangle. In this case, $\overrightarrow{B}$ is given by $\overrightarrow{G_\gamma L_p}$ and $\overrightarrow{T}$ is calculated according to the N_γ (*i.e* the normal of T_γ) and $\overrightarrow{B}$.

V_1, the opposite vertex to the adjacent side is projected in the plane of T_γ. We obtain $V_1^{'}$ and use it to compute texture coordinates of the hatching parameters H_1^γ (hatching parameters of V_1 for T_γ). This projection is applied to all vertices of T_c and all triangles T_α, T_β and T_γ.

As a first approach, we propose to blend the results given by these different texture coordinates producing a rendering whose aspect is continuous at the junction of triangles. Thus, we calculate a blending factor f for each vertex per adjacent triangle. This first approach is a compromise producing grayscale strokes (see on figure 4(b)). In practice, these artifacts are hardly visible on a 3D model as demonstrated in the figure 10 and in the addtional video (i.e the results section). Note that complete approach is palnned in future work.

As shown in figures 7 and 8, for each vertex V_i, we compute a contribution value that gradually decreases along the triangle. Depending on the dot product between two adjacency triangle normals we ensure that when two triangles have an angle less than $\frac{\pi}{2}$ radian, there is no contribution between them. For example in a cube, triangles of different faces should not influence each other. So, finally we obtain four hatching parameters per vertex of T_c.

We present below the algorithm 2 in which we compute for each emitted vertex V_i (*i.e* $i \in \{0,1,2\}$) of T_c its direction to the light $\overrightarrow{V_iL}$, the triangle normals $\overrightarrow{N_t}$ (*i.e* $t \in \{c,\alpha,\beta,\gamma\}$) and its hatching parameters H_i^t. These values will be used in the final step to render the model according to a lighting model.

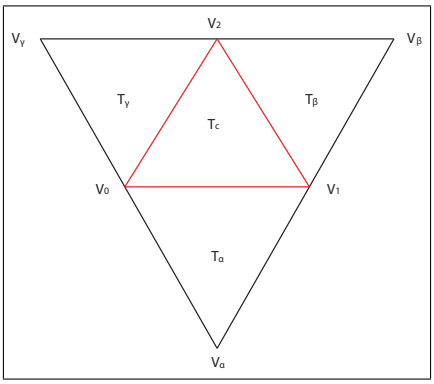


Figure 5: Triangle adjacency topology. The main triangle is T_c. Neighbor ones are indexed α, β, γ.

3.4 Tonal mapping

Our model is inspired both by the toon shading technique introduced in [Lak00] where the light intensity is given by a 1D texture and a Phong shading where contribution is computed per fragment.

This approach can be extended to 2D textures combining lighting to a palette of textures. In the fragment shader stage, by computing the dot product between fragment-light vector $\overrightarrow{FL}$ and the fragment nor-

foreach *triangle t in* $\{c, \alpha, \beta, \gamma\}$ **do**
  Compute normal and record it in $\overrightarrow{N_t}$;
  Compute projected light direction in T_t plane as previously described;
end
foreach *vertex i in* $\{0,1,2\}$ **do**
  foreach *triangle t in* $\{c, \alpha, \beta, \gamma\}$ **do**
    Compute V_i-light vector and record it in $\overrightarrow{V_iL}$;
    if $(t = c)$ **then**
      Compute texture coordinates of V_i as described in the previous subsection and record them in $H_i^c.st$;
      $H_i^c.f \leftarrow 1.0$;
      full contribution of T_c;
    else if V_i *does not belong to* T_t **then**
      $V_i^{'}$ is V_i projected in the T_t plane;
      Compute texture coordinates of $V_i^{'}$ in T_t and record them in $H_i^t.st$;
      $H_i^t.f \leftarrow -1.0$;
      no contribution of T_t;
    else
      Compute texture coordinates of V_i and record them in $H_i^t.st$;
      $H_i^t.f \leftarrow \overrightarrow{N_c} \cdot \overrightarrow{N_t}$;
      contribution depending on angle between T_c *and* T_t;
    end
  end
end

Algorithm 2: Compute Hatching parameters per Vertex according to adjacent triangles.

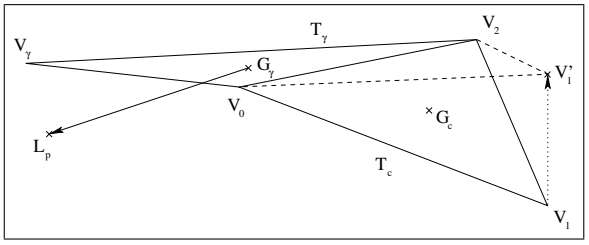


Figure 6: Texture coordinates obtained by adjacent triangle: used light direction and opposite vertex projection computation for continuity.

mal $\overrightarrow{N_t}$, we obtain, as shown in figure 9 the Lambertian term used to find the texture number in the TAM named *Tone*. Then, fragment information is automatically provided by linear interpolations in the GPU and according to each vertex information. Depending on the blending values $H_t.f$, the texture coordinates $H_t.st$, and the texture numbers *Tone*, we can compute, per fragment, the final color following the algorithm 3.

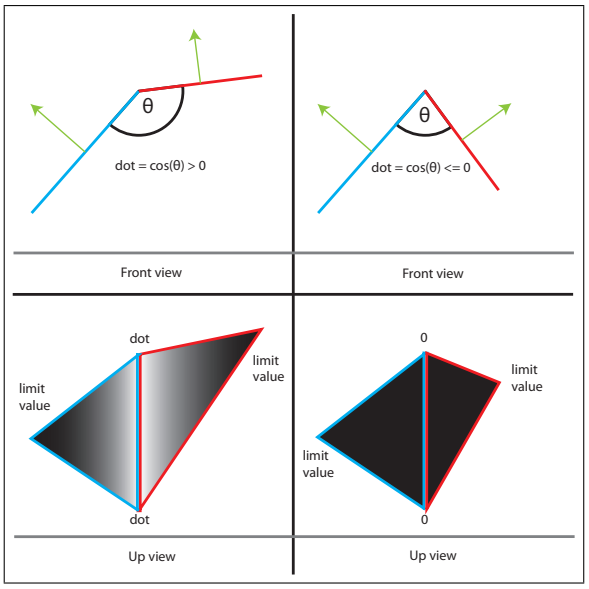


Figure 7: The blending value according to the angle between two adjacency triangles. Top of figure presents two front views of two triangles. On the left we study the case of $\cos\theta > 0$ and on the right we have the opposite case. Bottom of figure presents the blending factor computed per vertex and its interpolation along each triangle.

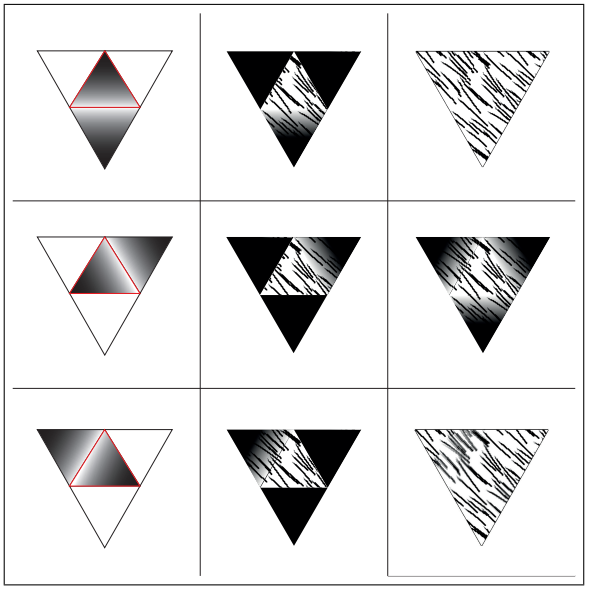


Figure 8: First column: contribution value between adjacent triangles. Second column: contribution applied to textures. Last column: result without blending, result of blending in T_c, result of blending in all triangles.

Note that, in our implementation, the level in the multiresolution TAM is chosen according to the fragment depth.

4 RESULTS

We present different results obtained with our model and discuss about performance, temporal and spatial

foreach *triangle t in* $\{c, \alpha, \beta, \gamma\}$ **do**
  Compute *Tone* according to $\overrightarrow{N_t}$ and $\overrightarrow{FL}$;
  Compute $Color_t$ determined by $H_t.st$ and *Tone*;
end
$FinalColor \leftarrow \sum(Color_t \times H_t.f) / \sum(H_t.f)$;
Algorithm 3: Compute the fragment color

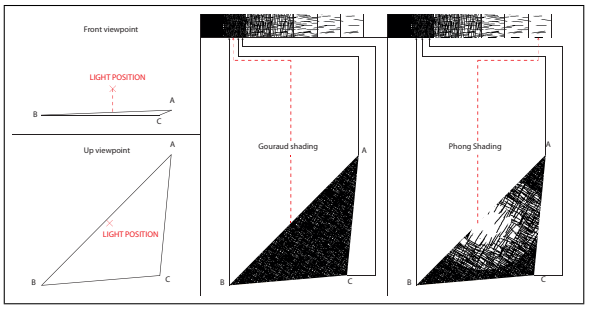


Figure 9: Example of texture number determining per fragment with different lighting techniques

coherence during scene animations.

Figure 10 presents an overview of possible results using our hatching model. The hatching parameters H_i^t of each vertex V_i can be globally modified on the fly, according to a matrix that provides the three basic transformations. Indeed, considering the $(\overrightarrow{T}, \overrightarrow{B}, \overrightarrow{N})$ coordinates system, we can shift $H_i^t(s,t)$ by adding a value in $[0;1]$ corresponding to a translation on $\overrightarrow{T}$ and/or $\overrightarrow{B}$ axis. We can scale each $H_i^t(s,t)$ using any scale value that modifies the texture repetition (see figure 10 second line). We can also modify $H_i^t(s,t)$ by making a rotation on the $\overrightarrow{N}$ axis to change the global texture orientation (see figure 10 first line and figure 14).

Our model provides a coherent lighting that follows the light and, as one can see on figure 11, reflects the fineness of the mesh.
Otherwise, our model gives the ability to represent different materials using different TAMs (see figure 12). TAMs can be used in addition to color materials (see figure 13 and figure 10).

Considering the performance aspect, our model is real time even for detailed geometry. Figure 15 illustrates our implementation performance expressed in number of frame per second considering different models. As one can see, for a 3D object composed by more than 500 000 triangles, our model produces renderings in 30 frames per second (NVIDIA Quadro FX 3800). As a comparison, we provide results both for our hatching model and for a basic fragment lighting model: a Phong shading. Note that, for both of them, we send the same geometry to the GPU including adjacency data. We obtain a ratio around 50% between these two renderings:

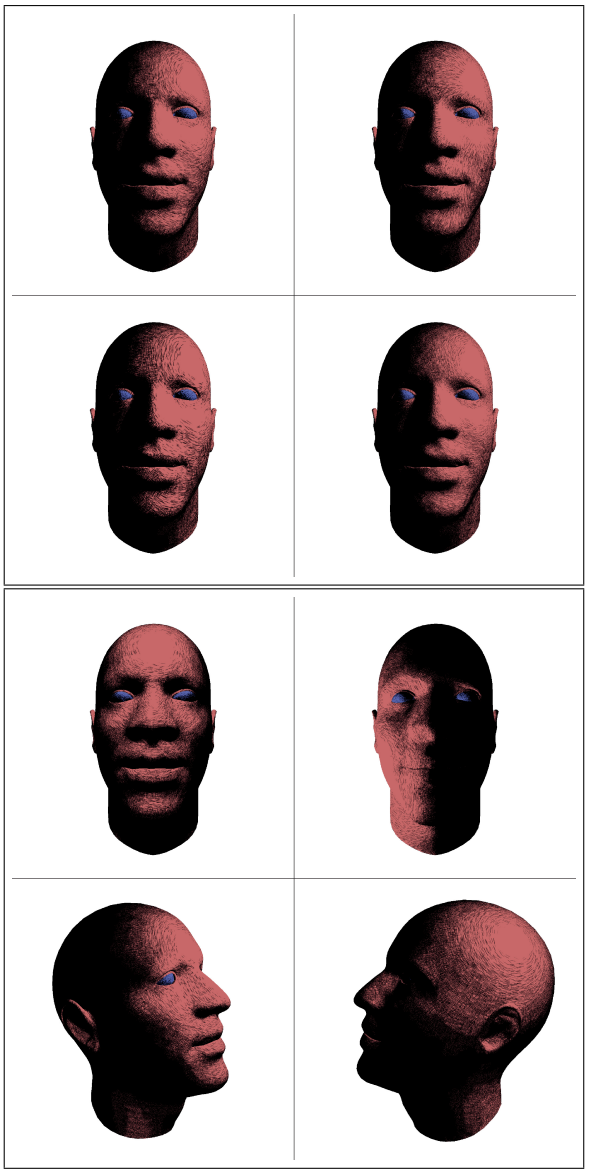


Figure 10: Results on 3D face model showing all different effects. First line: texture rotation along N axis for each triangle. Second line: different scale values are used. Third line: renderings using different light positions. Last line: model rotation for a fixed light.

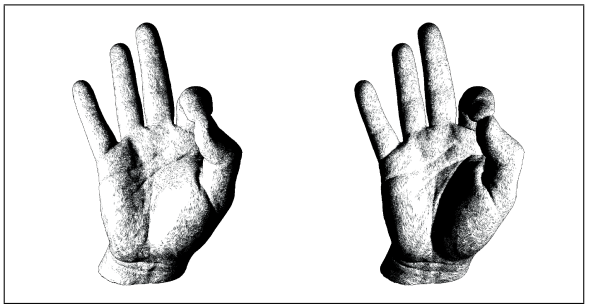


Figure 11: Results on the hand model with different light positions. Note that geometric details are always visible.

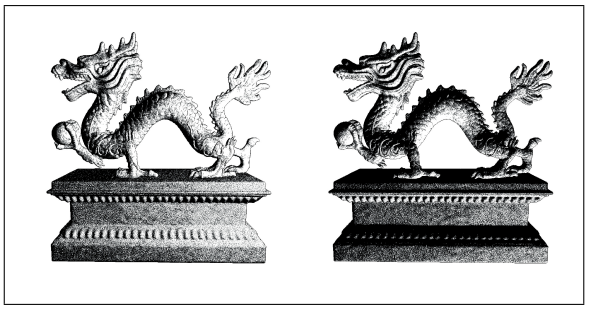


Figure 12: Results on the Stanford dragon model with different light positions and TAMs representing different materials.

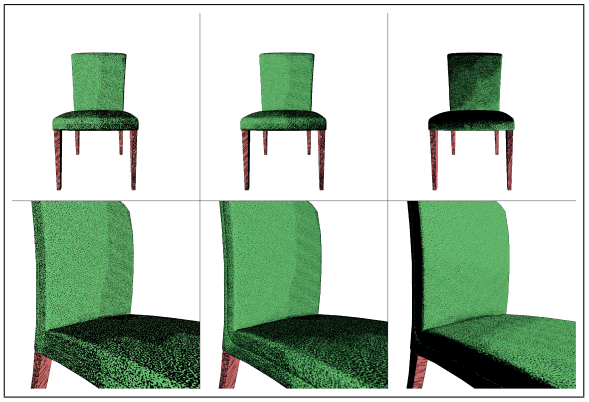


Figure 13: Results on a chair model with different light positions and TAMs + colorization representing distinct materials. First column shows results without adjacency blending.

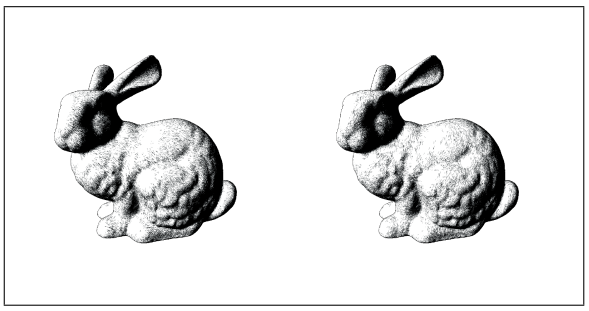


Figure 14: Results on the Stanford bunny model with different texture orientations. On the left, strokes are oriented toward the light. On the right, strokes are tangent to the light direction.

for a given geometry, our hatching rendering is twice slower than the Phong shading.

Concerning the spatial and temporal coherence of our model, a video showing our real time results is available at the following url:
`http://www.ai.univ-paris8.fr/~suarez`
We can notice that, between two consecutive frames, when lighting changes, variation of the selected TAM remains progressive while highlighting the object geometry.

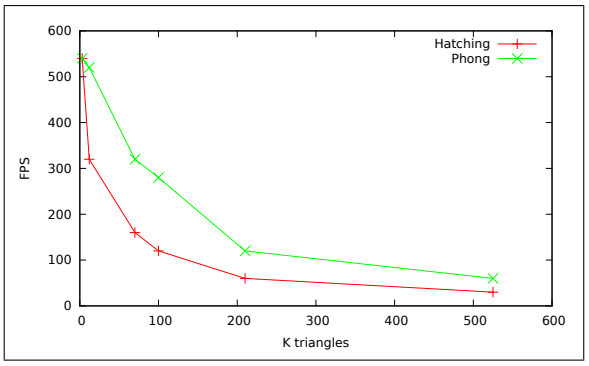


Figure 15: Graph showing performance of our rendering pipeline on different 3D models at a 1024x1024 rendering resolution (the used GPU is a Nvidia Quadro FX 3800). Results are expressed in FPS according to the number of triangles.

5 CONCLUSION

We have presented a model to produce hatching in 3D scene taking into account the brightness due to lighting, the orientation linked to the object geometry and the materials related to its texture. Our implementation is fully GPU and provides real time hatching on large scenes. Our model can be applied to any 3D models where the topology is constant. It provides hatching on static 3D models, animated 3D models and supports deformations. Triangle adjacency can be easily deduced from any 3D models at the loading step by indexing the model vertices. Moreover, no modeling engine modifications are needed. The model is also spacially and temporaly coherent since it gives continuous hatching during object animations and/or light displacements avoiding popping effects.

As future work, it will be interesting to have the ability to produce C_1-continuous strokes through adjacent faces. A procedural generation of TAMs constitutes an interesting way to adress this kind of problem. Moreover, the continuous aspect will be obtained automatically and dynamically without grayscale strokes. We also aim to manage multiple light sources in a single rendering pass by choosing a way that changes the hatching orientations (not simply blend them) according to these multiple sources. Finally, we plan to integrate drop and/or soft shadows and self-shadowing calculations to the model and then produce hatchings by disrupting orientations of faces affected by such shades.

6 REFERENCES

[Coc06] L. Coconu, O. Deussen, and H.-C. Hege. "Real-time pen-and-ink illustration of landscapes". In: *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pp. 27–35, ACM, New York, NY, USA, 2006.

[Deu00] O. Deussen and T. Strothotte. "Computer-generated pen-and-ink illustration of trees". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 13–18, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Hae90] P. Haeberli. "Paint by numbers: abstract image representations". In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 207–214, ACM, New York, NY, USA, 1990.

[Her00] A. Hertzmann and D. Zorin. "Illustrating smooth surfaces". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 517–526, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[Kap00] M. Kaplan, B. Gooch, and E. Cohen. "Interactive artistic rendering". In: *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pp. 67–74, ACM, New York, NY, USA, 2000.

[Lak00] A. Lake, C. Marshall, M. Harris, and M. Blackstein. "Stylized rendering techniques for scalable real-time 3D animation". In: *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pp. 13–20, ACM, New York, NY, USA, 2000.

[Pra01] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. "Real-time hatching". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 581–586, ACM, New York, NY, USA, 2001.

[Sai90] T. Saito and T. Takahashi. "Comprehensible rendering of 3-D shapes". In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 197–206, ACM, New York, NY, USA, 1990.

[Sal94] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin. "Interactive pen-and-ink illustration". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 101–108, ACM, New York, NY, USA, 1994.

[Sal97] M. P. Salisbury, M. T. Wong, J. F. Hughes, and D. H. Salesin. "Orientable textures for image-based pen-and-ink illustration". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 401–406, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[Web02] M. Webb, E. Praun, A. Finkelstein, and H. Hoppe. "Fine tone control in hardware hatching". In: *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pp. 53–59, ACM, New York, NY, USA, 2002.

[Win96] G. Winkenbach and D. H. Salesin. "Rendering parametric surfaces in pen and ink". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 469–476, ACM, New York, NY, USA, 1996.

A Muscle Model for Enhanced Character Skinning

Juan Ramos
Universidad Rey Juan Carlos
Modeling and Virtual Reality Group
C/ Tulipàn, s/n
28933, Mòstoles, Spain
Tomillo1235@gmail.com

Caroline Larboulette
University of South Brittany
IRISA-UBS Lab
Campus de Tohannic
56000, Vannes, France
Caroline.Larboulette@gmail.com

ABSTRACT

This paper presents a novel method for deforming the skin of 3*D* characters in real-time using an underlying set of muscles. We use a geometric model based on parametric curves to generate the various shapes of the muscles. Our new model includes tension under isometric and isotonic contractions, a volume preservation approximation as well as a visually accurate sliding movement of the skin over the muscles. The deformation of the skin is done in two steps: first, a skeleton subspace deformation is computed due to the bones movement; then, vertices displacements are added due to the deformation of the underlying muscles.
We have tested our algorithm with a GPU implementation. The basis of the parametric primitives that serve for the muscle shape definition is stored in a cache. For a given frame, the shape of each muscle as well as its associated skin displacement are defined by only the splines control points and the muscle's new length. The data structure to be sent to the GPU is thus small, avoiding the data transfer bottleneck between the CPU and the GPU. Our technique is suitable for applications where accurate skin deformation is desired as well as video games or virtual environments where fast computation is necessary.

Keywords

Muscle, Skinning, Character Animation, Real-Time, GPU

1 INTRODUCTION

While the rendering of static scenes can now be so realistic that it is very often difficult to distinguish virtual images from real photographs, the same isn't true for animation, especially for characters. Despite the tremendous progress that has been made, anybody can tell the difference between a video of a real person in movement and a 3*D* character animation. This is because a character is a complex object composed of many different parts in interaction and a character is familiar to human beings which leaves little room for inaccuracy.

It is common practice to decompose the animation of a character into two problems: the animation of the skeleton and the animation of what's around it. In this paper, we address the second one. By describing the individual layers composing an object rather than just its surface, we believe that it is possible to generate more accurate deformations of the outside skin layer.

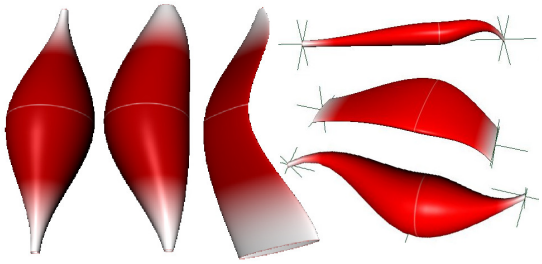


Figure 1: Various muscle shapes generated with our Maya plugin. From left to right: original template shape; non-linear sweeping spline; flat muscle; arbitrary muscles. The white line shows the position of the muscle belly center **C**.

Anatomically based models have first been introduced to Computer Graphics in 1992 by Chen and Zeltzer [Che92a] who modeled an accurate frog's muscle using finite elements. Subsequent simpler models have then been used with the sole goal of enhancing 3*D* characters' skin deformation through the animation of individual muscles. In contrary to what happens in real life, they deform as a response to bone deformation and their only purpose is to obtain visually more appealing deformations. Although that approach has been used for movie production (Shrek, Lord of the Rings [Gra]) and is included in recent versions of

Autodesk Maya Software [May13a] it has remained computationally too expensive and too difficult to set up for a wider use including video games or virtual environments. We propose a new muscle model that addresses both problems.

We do not intend to model dynamic effects or wrinkles that can be added as additional layers [Cap02a, Lar04a]. We do however provide a *tension* parameter that may be used to control a soft tissues dynamics simulation such as [Jam02a, Lar05a]. These additional layers are outside the scope of this paper that aims at presenting the muscle model and it's GPU implementation.

Our muscle model is defined by a few parameters that allow artists to express complicated shapes (see figure 1). Although it is best suited for fusiform and multipennate (broad origins and insertions) muscle shapes, multi-belly muscles such as the *pectoralis mayor* or the *trapezius* can be generated by grouping several fusiform shapes like in [Sch97a].

Animating muscle deformation as a response to bone movement has the advantage that it resembles the way artists work, but this implies that skin deformation due to the tension of the underlying muscles is not taken into account. To overcome this problem, we introduce a muscle *tension* parameter allowing the muscle to change the shape of its cross-section while preserving its overall volume. In addition, we achieve a sliding behavior of the skin over the muscles by restricting the displacement of each skin vertex to directions perpendicular to the muscles' fibers which visually improves the results obtained by previous geometric models where the skin vertices were rigidly attached to the muscles primitives.

To address the challenges of accurately animating anatomically based character models in real-time, we introduce (1) a new parametric fusiform muscle model capable of both isometric and isotonic contraction that preserves its overall volume (section 3), (2) a new geometric algorithm for skin deformation that achieves skin sliding behavior without the high cost of a physically based simulation (section 4), and (3) a GPU implementation of our model including normal correction that is suitable for use in video games (section 5). We achieve frame rates of 25 Hz on a dual core 3.0 GHz Pentium 4 PC equipped with a GeForce 8800 GTX graphics board for 25 instances of a mesh composed of 82000 triangles and 56 muscles, which represents a computational overhead of $20-25\%$ compared to a smooth skinning only.

2 RELATED WORK

The modeling and animation of individual muscles has been an extensive topic of research for a few decades. For a comprehensive state-of-the-art, the reader may refer to [Lee12a].

Previous work range from anatomically accurate models to models only vaguely faithful to the real anatomy. On the one hand, we find the muscle models that serve as a mean to move the bones of the skeleton. Because they are difficult to control in their actual state, those models are still marginal although they continue to receive some attention [Lee06a]. A complete biomechanical model of the upper body can be found in [Lee09a]. On the other hand, we find techniques that try to mimic some muscle deformation of the skin although either no individual muscle is defined [Wan02a, Moh03a, Cap07a] or the muscles do not correspond to real ones [Pra05a]. The skinning based techniques give nice results provided the user designs accurate input shapes [Wan02a]. However, with only anisotropic scaling [Cha89a, Wan02a, Moh03a] along X, Y and Z-axis, it is very difficult to obtain realistic muscle deformations. [Wan07a, Web07a, San08a] present data driven techniques for synthesizing skin deformations from skeletal motion. While the results produced are definitely more accurate, those techniques require a set of examples and thus cannot be applied to characters that need exaggerated deformations or for movements the actor did not perform or the animator did not create.

The alternative is to define individual muscles that serve as a mean to deform the outer skin layer but do not control the bones and the skeleton movement like in reality. Our work fits into this paradigm. In 1997, Wilhelms and Scheepers et al. [Wil97a, Sch97a] both proposed a first generic muscle model based on ellipsoids. While the results were visually appealing at that time, there were several limitations to the model. Firstly, the shape of the muscle belly and of the tendons was constrained to elliptical ones which made it hard to cover all of the body muscles, and even the *biceps* and *triceps* of the arm could not be correctly approximated. Subsequent parametric models [Wil97b, Lee07a] tried to solve the setup problem. Parametric muscles that have been designed for one character are easy to re-use onto another character as only a few scalings need to be done. Our muscle model is also based on the idea that only a small set of parameters is sufficient to create a wide variety of shapes. Moreover, unlike [Lee07a], our parameterization is suitable for a GPU implementation.

More recently, physically based models of muscles have been investigated. Whether they are based on mass-spring systems [Ned00a, Aub00a], finite elements methods [Zhu98a] or finite volume methods [Ter03a, Ter05a] they are difficult to set up and several orders of magnitude slower than the model we propose. In addition, the muscle shape is defined by either a polygonal mesh or a volumetric decomposition in tetrahedra which is totally unsuitable for the modern GPU pipelines.

To deform the skin and obtain a sliding effect of the skin over the muscles, [Wil97b] uses masses and springs to model the skin as well as to anchor the skin to the underlying muscles and bones. While this gives pleasing results, this step is very heavy in terms of setup (stiffness and damping coefficients) and computation. Turner et al. [Tur93a] use a physically based simulation of elasticity to deform an elastic skin. We avoid the heavy cost of a physically based simulation by proposing a new geometric algorithm to displace the vertices of the skin as a response to the muscles deformation, which allows sliding effects. This is an improvement compared to other geometric models where the skin is rigidly anchored to underlying components [Wil97b, Lee07a].

3 OUR MUSCLE MODEL

Similar to previous models [Wil97b, Ned00a, Aub00a, Lee07a], we make the assumption that the force exerted by the muscle follows a fictive *action line* $A(t)$ that lies on the main axis of the muscle shape. This action line is anchored to the bones at the *origin* and *insertion* points, thus elongating and shortening as the skeleton moves. The muscle belly is attached through deformable tendons at both ends.

Overview The general shape of our muscle model is a modified generalized cylinder defined by the sweep of a *thickness curve* $C_T(t)$ along a *sweeping curve* $C_S(t)$. $C_S(t)$ is a three-dimensional spline representing the profile of the muscle. It is defined by two $3D$ points: the origin $\mathbf{O}$ and insertion $\mathbf{I}$. $C_T(t)$ is a one-dimensional function representing the muscle's thickness as a function of t. The muscle's cross-section at t is thus defined as an ellipse placed along $C_S(t)$ whose thickness is given by $C_T(t)$ and a scaling factor $\mathbf{s_f}$. $\mathbf{s_f}$ is applied in a user-chosen direction $\overrightarrow{\mathbf{S_d}}$ perpendicular to the action line. This scaling is only applied to the sections, not to the spline curves. See figure 2 for a sketch.

To avoid intersections between sections and to simplify the volume conservation computation detailed in section 3.2.1, the sections do not follow the curvature of $C_S(t)$ but are always perpendicular to the action line $A(t)$.

Modeling We have implemented a Maya plugin that allows the user to modify the curves control points, the scaling factor $\mathbf{s_f}$ as well as the scaling direction $\overrightarrow{\mathbf{S_d}}$. As we will see in section 3.1, the control points are constrained and cannot be freely moved, which reduces the number of variables to control. In addition, the muscle belly center $\mathbf{C}$ can be displaced along the segment $\mathbf{OI}$. It is thus possible to obtain a wide variety of asymmetrical shapes. See figure 1 for examples.

3.1 Shape Computation

$C_S(t)$ and $C_T(t)$ Bézier curves defining the muscle shape are each composed of two cubic spline segments

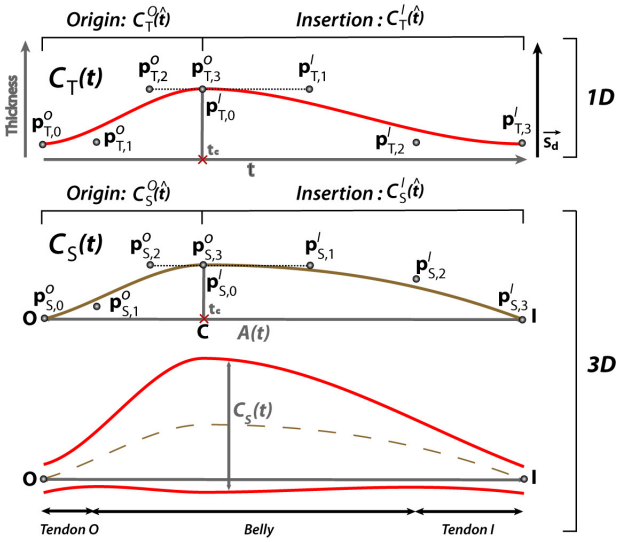


Figure 2: Top: *thickness curve* $C_T(t)$; center: *sweeping curve* $C_S(t)$; bottom: muscle's longitudinal section resulting from the sweeping of $C_T(t)$ along $C_S(t)$ with $C_S(t)$ represented in dashed line.

$C_S^O(\hat{t}), C_S^I(\hat{t})$ and $C_T^O(\hat{t}), C_T^I(\hat{t})$ respectively called *origin* and *insertion segments*, joined with $\mathscr{G}^1$ continuity. Mapping between t and $\hat{t}$ is explained later. $C_T(t)$ is a piecewise one-dimensional function which represents the varying thickness of the muscle at a parametric point $t \in [0,1]$ along $C_S(t)$.

Each Bézier cubic spline segment is defined by four control points $\mathbf{p}_j$, $j \in [0,3]$ and the cubic Bernstein polynomials $B_j^3(\hat{t})$ by:

$$\begin{aligned} C(\hat{t}) &= \textstyle\sum_{j=0}^{3} \mathbf{p}_j B_j^3(\hat{t}) \\ &= (1-\hat{t})^3\mathbf{p}_0 + 3\hat{t}(1-\hat{t})^2\mathbf{p}_1 + \\ &\quad 3\hat{t}^2(1-\hat{t})\mathbf{p}_2 + \hat{t}^3\mathbf{p}_3 \end{aligned} \qquad (1)$$

Henceforward we will write $\mathbf{p}_{0-3}^O$ and $\mathbf{p}_{0-3}^I$ the four control points of the *origin* and *insertion* spline segments respectively, regardless of their dimension.

$\mathscr{G}^1$ continuity is obtained by merging $\mathbf{p}_3^O$ and $\mathbf{p}_0^I$ and aligning $\mathbf{p}_2^O$, $\mathbf{p}_3^O$ and $\mathbf{p}_1^I$ in the cubic spline segments for both the *thickness* and the *sweeping* curves (see figure 2). $\mathbf{p}_0^O$, $\mathbf{p}_1^O$, $\mathbf{p}_2^I$ and $\mathbf{p}_3^I$ can take any arbitrary values chosen by the user.

The action line $A(t)$ defined by the origin $\mathbf{O}$ and the insertion $\mathbf{I}$ of the muscle can be expressed by the following parametric equation:

$$A(t) = \mathbf{O} + t(\mathbf{I}-\mathbf{O}) \quad t \in [0,1] \qquad (2)$$

Let $\overrightarrow{\mathbf{S}}$ be the unit vector perpendicular to both $\overrightarrow{\mathbf{S_d}}$ and $A(t)$:

$$\overrightarrow{\mathbf{S}} = \frac{\overrightarrow{\mathbf{OI}} \times \overrightarrow{\mathbf{S_d}}}{\|\overrightarrow{\mathbf{OI}} \times \overrightarrow{\mathbf{S_d}}\|} \qquad (3)$$

The orthogonal basis $\mathscr{B} = \{\overrightarrow{\mathbf{S}}; \overrightarrow{\mathbf{S_d}}; \overrightarrow{\mathbf{OI}}\}$ defines the *section space*. Each of $C_S(t)$ control points, $\mathbf{p}^O_{S,0-3}$ and $\mathbf{p}^I_{S,0-3}$, can be defined in *section space* by projecting them onto $\mathscr{B}$ orthogonal vectors.

Efficient skin vertices displacement as a response to muscles deformation such as described in section 4.2 relies on having a direct mapping of the parametric space of the action line onto the parametric space of the spline segments. This is simply obtained by making local Bézier parameter $\hat{t}$ linearly dependent on t. Thanks to the linear precision property of the Bézier spline, uniformly distributed control points on a straight line produce that straight line:

$$\sum_{j=0}^{n} \frac{j}{n} B_j^n(t) = t \tag{4}$$

In our case $n = 3$. The control points $\mathbf{p}^O_{S,0-3}$ on $C^O_S(t)$ are spaced by $\|\overrightarrow{\mathbf{OC}}\|/3$ along its t coordinate and the control points $\mathbf{p}^I_{S,0-3}$ on $C^I_S(t)$ are spaced by $\|\overrightarrow{\mathbf{CI}}\|/3$. In *section space*, $\overrightarrow{\mathbf{OI}}$ coordinate is t, so $\mathbf{p}^O_{S,0-3}$ and $\mathbf{p}^I_{S,0-3}$ positions can be expressed with only two coordinates plus t.

Let $\mathbf{C} = A(t_c)$ be the muscle belly center such as $t_c = \|\overrightarrow{\mathbf{OC}}\|/\|\overrightarrow{\mathbf{OI}}\|$. The local $\hat{t}$ is obtained by:

$$\hat{t} = \begin{cases} t/t_c & \text{if } t <= t_c, \\ (t-t_c)/(1-t_c) & \text{if } t > t_c. \end{cases} \tag{5}$$

and used in $C^O_S(\hat{t})$, $C^O_T(\hat{t})$ and $C^I_S(\hat{t})$, $C^I_T(\hat{t})$ respectively to define the shape of the muscle as a function of t along the action line.

3.2 Muscle Deformation

Muscle is an incompressible, anisotropic, hyper-elastic material that undergoes two types of contractions: isotonic and isometric. In both cases, the overall volume of the muscle remains roughly constant [Wil97b].

In the case of an *isotonic* contraction, the bones move but the tension of the muscle remains constant. The change in shape is thus only due to the shortening/elongation of the muscle. If it shortens, the belly of the muscle bulges. It decreases in the other case. The action line, anchored to the bones at **O** and **I**, changes length as the skeleton moves. In order to preserve the volume of the muscle, we recompute the new belly thickness while keeping tendons thicknesses constant. This is detailed in section 3.2.1.

In the case of an *isometric* contraction, the bones do not move and the muscle length doesn't change. However, the shape of the muscle is modified. When the tension increases, the muscle bulges in one direction while narrowing in the perpendicular dimension. Tendons also bulge in the same direction and become more visible. The muscle and the tendons all relax to the rest state when the tension decreases. Again, the overall volume of the muscle remains constant. This is achieved through a tension parameter **tension** that allows the muscle to scale in the direction $\overrightarrow{\mathbf{S_d}}$ chosen by the artist. The inverse of that scale is applied in the perpendicular direction $\overrightarrow{\mathbf{S}}$. This is detailed in section 3.2.2.

In real life, both types of contraction happen at the same time. First, the tension of the muscle increases. When it reaches a certain threshold, the bones start moving while the tension decreases until the bones reach the new position. By moving the bones and varying the tension parameter of our model, it is possible to achieve this natural behavior.

3.2.1 Volume Preservation

As muscle sections are parallel to each other along the sweeping curve, two consecutive sections form the elliptic bases of an oblique truncated cone of height the distance between the two sections. The volume of the muscle is thus equal to the sum of an infinite number of such cones of infinitely small height. The volume of an oblique truncated cone is the same as the one of a right cone, therefore we can compute the volume as if $C_S(t)$ were straight and equal to $A(t)$.

Cone volume computation is costly and we have found that a very coarse discretization of the muscle's volume in only two truncated elliptic cones was a good and very stable approximation (see figure 3). Indeed, it is more important to keep the differential volume error low between the different muscle states (elongated, rest, compressed) rather than having an exact volume. For the muscles we used in our demos, we have measured an error of 10% in average with respect to the exact volume computation but only a 1% error deviation between extreme muscle states (150% elongated, 50% shortened). This means that the volume remains constant with an error of only 1%, which is negligible.

Let (a_X, b_X) be the axes of the three ellipses defining the two cones at the origin **O**, center **C**, and insertion **I** points. The a_X axes are aligned with the scaling direction $\overrightarrow{\mathbf{S_d}}$ and their length is $\mathbf{a}_X$. b_X axes are perpendicular to the a_X ones and their lengths $\mathbf{b}_X$ are defined as $\mathbf{b}_O = \mathbf{p}^O_{T,0}$, $\mathbf{b}_C = \mathbf{p}^O_{T,3} = \mathbf{p}^I_{T,O}$ and $\mathbf{b}_I = \mathbf{p}^I_{T,3}$. The ratios $\mathbf{a}_X/\mathbf{b}_X = \mathbf{s_f}$.

The total volume of the muscle is computed by equation (6) which, divided by $\mathbf{s_f}$, results in equation (7).

$$\begin{aligned} V &= \|\overrightarrow{\mathbf{OC}}\|\pi\left(\tfrac{2\mathbf{a}_C\mathbf{b}_C+\mathbf{a}_C\mathbf{b}_O+\mathbf{a}_O\mathbf{b}_C+2\mathbf{a}_O\mathbf{b}_O}{6}\right) \\ &+ \|\overrightarrow{\mathbf{CI}}\|\pi\left(\tfrac{2\mathbf{a}_C\mathbf{b}_C+\mathbf{a}_C\mathbf{b}_I+\mathbf{a}_I\mathbf{b}_C+2\mathbf{a}_I\mathbf{b}_I}{6}\right) \end{aligned} \tag{6}$$

$$\begin{aligned} V &= \tfrac{\pi\mathbf{s_f}}{3}(\|\overrightarrow{\mathbf{OC}}\|(\mathbf{b}_C{}^2 + \mathbf{b}_O{}^2 + \mathbf{b}_C\mathbf{b}_O) \\ &+ \|\overrightarrow{\mathbf{CI}}\|(\mathbf{b}_C{}^2 + \mathbf{b}_I{}^2 + \mathbf{b}_C\mathbf{b}_I)) \end{aligned} \tag{7}$$

The initial volume is computed at rest state. To keep a constant volume as well as a constant thickness for the tendons (desired feature for muscles deforming under isotonic contraction), $\mathbf{b}_C$ needs to be recomputed at each timestep. From (7), we obtain a quadratic equation $A{\mathbf{b}_C}^2 + B\mathbf{b}_C + C = 0$ with:

$$\begin{aligned} A &= \|\overrightarrow{\mathbf{OI}}\| \\ B &= \|\overrightarrow{\mathbf{OC}}\|\mathbf{b}_O + \|\overrightarrow{\mathbf{CI}}\|\mathbf{b}_I \\ C &= \|\overrightarrow{\mathbf{OC}}\|{\mathbf{b}_O}^2 + \|\overrightarrow{\mathbf{CI}}\|{\mathbf{b}_I}^2 - \frac{3V_{initial}}{\pi \mathbf{s_f}} \end{aligned} \tag{8}$$

It can be solved by taking its positive solution ${\mathbf{b}_C}' = \frac{-B+\sqrt{\delta}}{2A}$ with $\delta = B^2 - 4AC$.

As mentioned in section 3.1, to keep the belly shape continuous, $\mathbf{p}^O_{T,2}$, $\mathbf{p}^O_{T,3}$ and $\mathbf{p}^I_{T,1}$ must be kept collinear as $\mathbf{b}_C$ changes and the slope of segment $\mathbf{p}^O_{T,2}\mathbf{p}^I_{T,1}$ must also be kept constant.

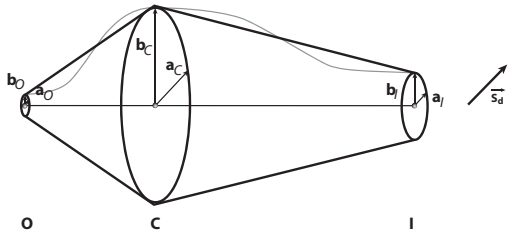


Figure 3: The volume of the muscle is approximated by the volume of two elliptic cones defined by $(\mathbf{O}, \mathbf{a}_O, \mathbf{b}_O)$, $(\mathbf{C}, \mathbf{a}_C, \mathbf{b}_C)$ and $(\mathbf{C}, \mathbf{a}_C, \mathbf{b}_C)$, $(\mathbf{I}, \mathbf{a}_I, \mathbf{b}_I)$.

3.2.2 Tension Parameter

From equation (7) it is straightforward that to maintain a constant volume, the scaling factor $\mathbf{s_f}$ should be kept constant. As $\mathbf{s_f} = \mathbf{a}_X/\mathbf{b}_X$, this can be achieved by multiplying axes lengths $\mathbf{a}_X$ by **tension** while dividing axes lengths $\mathbf{b}_X$ by the same amount at the origin, center and insertion points.

The new axes lengths $\tilde{\mathbf{a}}_X$ and $\tilde{\mathbf{b}}_X$ when the muscle is under isometric contraction can thus be expressed as follows:

$$\begin{aligned} \tilde{\mathbf{a}}_X &= \mathbf{a}_X * \mathbf{tension} \\ \tilde{\mathbf{b}}_X &= \mathbf{b}_X / \mathbf{tension} \end{aligned} \tag{9}$$

Figure 4 shows a muscle in isometric contraction for different values of the tension parameter. At rest state (left), **tension** $= 1$. The muscle is under tension for **tension** > 1, and, for increasing values of **tension**, the muscle bulge is more noticeable (middle and right).

At any time during animation, both types of contraction are involved. The volume of the muscle is preserved by first applying equation (7) that accounts for changes in length of the action line, then equation (9) that takes the strain into account.

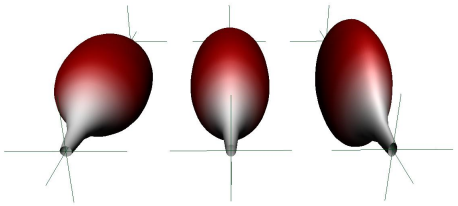


Figure 4: Varying tension parameter. From left to right: **tension** $= 1$, **tension** $= 1.15$ and **tension** $= 1.3$.

4 SKIN DEFORMATION & RENDERING

The skin/clothes layer is the only visible layer on a character. The muscles serve as an underlying tool to help deform the skin in a more accurate way. We use a two-steps approach. First, the skin layer is modified using a linear blend skinning as a response to the skeleton's movement. Note that other skinning techniques [Kav05a, Kav07a, Kav08a] can be used. The skin layer is subsequently deformed as a response to the muscles deformation by applying a displacement to each influenced vertex. We perform this step on the GPU.

The way a muscle deforms the surrounding skin is crucial to obtain good results. As the relationship between the skin and the muscle can be very different depending on the part of the body affected, we introduce a weighted deformation scheme as well as a fat layer that allow the animator to adjust the skin deformation. The fat layer is in charge of keeping a distance between the skin and the underlying muscle and more or less attenuates the deformation due to the muscle bulging depending on the chosen weighting system (section 4.1).

To achieve a skin sliding behavior over the muscles, we propose a vector oriented vertices displacement (section 4.2).

4.1 Muscle Influence

A skin vertex is influenced by a muscle if it lies in its neighborhood. This influence together with an associated weight are evaluated at the bind pose. In existing commercial programs or muscle-dedicated plugins the user paints the vertices influences by hand for each muscle. We provide an automatic way to assign normalized weights varying from 0 to 1 to skin vertices, a weight of 0 meaning no influence at all, and 1 meaning full muscle influence.

The influence computation is achieved by projecting perpendicularly each vertex of the concerned limb $\mathbf{V_i}$ onto the action line $A(t)$ of each muscle. If the projection point lies outside **OI**, then this particular vertex is not influenced by the muscle. In the other case the muscle's influence depends on a neighborhood template assigned by the user.

We provide three neighborhood templates:

Radial template: all of the vertices that lie within a given distance from the action line are influenced.

Half-space template: all of the vertices that lie on the upper side of the plane defined by the action line and having $\overrightarrow{\mathbf{S_d}}$ as a normal are influenced.

Fitted template: similar to the half-space template but with the additional restriction that the ray $(\mathbf{V_i}, -\overrightarrow{\mathbf{S_d}})$ must intersect the muscle shape for the vertex $\mathbf{V_i}$ to be influenced.

The user chooses a template depending on how he wishes the skin to be influenced by the muscle. Examples of the three templates are shown on figure 5, top row, for a given muscle.

If the vertex $\mathbf{V_i}$ is influenced by the muscle, its weight $\mathbf{w_i}$ is obtained depending on an automatic weighting system chosen by the user. We have implemented two of them:

Binary weighting system: the weight $\mathbf{w_i}$ of $\mathbf{V_i}$ is set to 1 if the vertex $\mathbf{V_i}$ is influenced by the muscle and 0 in the other case.

Spline-based weighting system: the weight is obtained by dividing the muscle thickness $C_T(t)$ at $\mathbf{V_i}$ by $max(\mathbf{p}^O_{T,2}, \mathbf{p}^O_{T,3}, \mathbf{p}^I_{T,1})$. We obtain weights $\mathbf{w_i} \in [0,1]$ that depend on the local thickness of the muscle.

These weighting systems must be used depending on the expected deformation of the muscle on the skin (see figure 5, bottom row). The first one is indicated for very fitted zones with almost no fat, so the tendons can be seen as well as almost the whole shape of the muscle. Note that the skin deformation is continuous and smooth because the muscle shape is $\mathscr{G}^1$ continuous and the skin follows that shape. The second one attenuates the deformation of the skin with respect to the previous one, and is indicated for fatty zones were only a small bulge due to the muscle is expected. The computed weights are also smooth, so the deformation remains continuous.

Figure 5: From left to right: *radial* influence template, *half-space* template, and *fitted* template. Top row shows the corresponding *binary* weights while bottom row shows the *spline-based* weights.

4.2 Skin Vertices Displacement

Vector oriented displacement makes the skin slide over the muscles by preserving a distance between them without rigidly anchoring the skin vertices to the muscles' primitives. To explain the deformation technique we first suppose that there is no fat layer and that all vertices have a weight of 1. It results in the skin being displaced to fit the muscles surfaces. How the fat layer and weights affect the vertices displacement is explained in section 4.3.

Let $\mathbf{V_i}$ be the current skin vertex we want to displace onto the muscle's surface $\mathbf{V'_i}$. This vertex is displaced in $\overrightarrow{\mathbf{S_d}}$ direction defined by the user as the main direction of bulging.

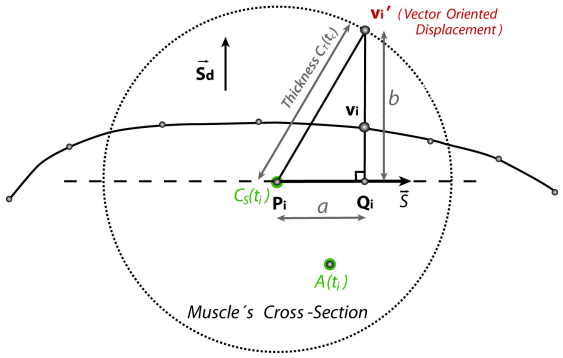


Figure 6: Skin vertices displacement algorithm: $\mathbf{V_i}$ is displaced in order to lie on the muscle's surface at $\mathbf{V'_i}$.

The algorithm works as follows (see figure 6): we first compute $\mathbf{P_i} = C_S(t_i)$ as the central point of a muscle section on $C_S(t)$ curve. The muscle's section on which we solve the intersection now contains $\mathbf{P_i}$ and $\mathbf{V_i}$. Let $C_T(t_i)$ be the thickness of the muscle at the point $\mathbf{P_i}$ and $\overrightarrow{\mathbf{S}}$ be the unit vector perpendicular to both the action line $A(t)$ and the displacement vector $\overrightarrow{\mathbf{S_d}}$ (see equation (3)).

The point $\mathbf{Q_i}$ is defined as the projection of $\mathbf{V_i}$ onto $\overrightarrow{\mathbf{S}}$. The distance from $\mathbf{P_i}$ to $\mathbf{Q_i}$ can simply be expressed as $\mathbf{a} = \overrightarrow{\mathbf{P_iV_i}} \cdot \overrightarrow{\mathbf{S}}$. Because $\mathbf{a}$ and $\mathbf{b}$ (see figure 6) form a right triangle, the new position of vertex $\mathbf{V_i}$ is given by the two following equations:

$$\mathbf{b} = \sqrt{C_T(t_c)^2 - \mathbf{a}^2} \tag{10}$$

$$\mathbf{V'_i} = \mathbf{Q_i} + \overrightarrow{\mathbf{S_d}}\,\mathbf{b} \tag{11}$$

At this point we have found $\mathbf{V'_i}$ for a circular muscle section. To make the algorithm work with elliptical sections we only have to multiply $\mathbf{b}$ by $\mathbf{s_f}$ and **tension**.

$$\mathbf{V'_i} = \mathbf{Q_i} + \overrightarrow{\mathbf{S_d}} \cdot \mathbf{b} \cdot \mathbf{s_f} \cdot \mathbf{tension} \tag{12}$$

4.3 Fat Layer and Weighted Deformation

We have introduced vertex weights $\mathbf{w_i} \in [0,1]$ in section 4.1 to attenuate the effect of muscles on the deformation of the skin. In the case where no fat layer

exists, those weights directly influence the vertex displacement $||\overrightarrow{\mathbf{V_i}\mathbf{V_i'}}||$. The new vertex position $\mathbf{V_i''}$ is thus given by:

$$\mathbf{V_i''} = \mathbf{V_i} + \overrightarrow{\mathbf{V_i}\mathbf{V_i'}}\ \mathbf{w_i} \tag{13}$$

However, muscles can deform the skin through a fat layer. Our implementation of the fat layer consists in calculating a distance $\mathbf{L_{fat}}$ between the muscle and the skin at bind pose and keeping it constant during animation. To keep distance consistency $\mathbf{L_{fat}}$ must measure vector oriented distance to the muscle.

In that case, the weights $\mathbf{w_i}$ are applied to the fat layer offset $\mathbf{L_{fat}}$ and the new vertex position $\mathbf{V_i''}$ is given by:

$$\mathbf{V_i''} = \mathbf{V_i} + \frac{\overrightarrow{\mathbf{V_i}\mathbf{V_i'}}}{||\overrightarrow{\mathbf{V_i}\mathbf{V_i'}}||}\ (||\overrightarrow{\mathbf{V_i}\mathbf{V_i'}}|| + \mathbf{L_{fat}}\ \mathbf{w_i}) \tag{14}$$

4.4 Normal Correction

Visual appearance of 3D characters is dramatically improved by using several textures for diffuse lighting, bump mapping, and specular mapping. To apply those algorithms, it is necessary to have the normal and binormal vectors for each mesh vertex. When a muscle displaces a vertex $\mathbf{V_i}$ into a new position, the normal $\overrightarrow{\mathbf{N_i}}$ obtained from the previously applied bone skinning technique must be recomputed (see figure 7).

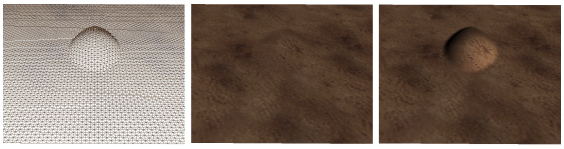


Figure 7: Left: displaced wireframe mesh; center: same mesh with texture applied but no normal correction; right: same mesh with texture and normal correction.

As the normals of the muscle shape can't be directly mapped to the skin due to the fat layer and influence weighting, the ideal solution is to recompute the skin normals from vertex adjacency data. Current GPUs only have edge adjacency available at the input of the pipeline, so we have used a pre-calculated buffer that contains the adjacency information for each vertex.

The character deformation is divided into two passes: the first pass computes the mesh deformation due to the skinning and the muscles; and the second pass renormalizes the normals by looking up in the adjacency buffer and computing regular per vertex normals. We need to recalculate the normals for the entire mesh as skeleton subspace transformed normals are different (see figure 8).

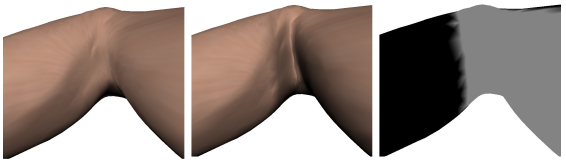


Figure 8: Arm example without muscles. Left: skeleton subspace normals calculation for the whole mesh; center: a discontinuity can be seen when using different normal calculation algorithms; right: normal calculation mask used in the central picture: in black, GPU skeleton subspace normals calculation; in grey, renormalization by computing normals per vertex.

5 GPU IMPLEMENTATION AND RESULTS

We have implemented the vertices displacements due to muscle deformation on a vertex program on the GPU. Two key points make our implementation run in real-time: the use of a parametric muscle model that needs to send little data to the GPU per frame, and the discretization of the Bézier spline basis to avoid high computational cost.

5.1 CPU-GPU data transfer

Muscle volume preserving and movement due to the skeleton animation is calculated on the CPU while the vertex bone skinning, muscle displacement and normal correction are computed on the GPU. Table 1 lists the data sent per animation timestep to the GPU for each muscle. While some of this data is redundant for reducing the amount of computation on the GPU, the total size is of only 10 vectors per muscle per frame.

Type	Information	Symbol	Size (bytes)
float3	origin	$\mathbf{O}$	12
float4	unit side vector center point	$\overrightarrow{\mathbf{S}}$ $\mathbf{C}$	16
float4	action line and its size	$\overrightarrow{\mathbf{OI}}$, $\|\|\overrightarrow{\mathbf{OI}}\|\|$	16
float4	unit scale direction, scale amount	$\overrightarrow{\mathbf{S_d}}$, $\mathbf{s_f}$	16
float4	C_T^O control points	$\mathbf{p}_{T,0-3}^O$	16
float4	C_T^I control points	$\mathbf{p}_{T,0-3}^I$	16
float4	C_S^O control points X	$\mathbf{p}_{S,0-3}^O$	16
float4	C_S^O control points Y	$\mathbf{p}_{S,0-3}^O$	16
float4	C_S^I control points X	$\mathbf{p}_{S,0-3}^I$	16
float4	C_S^I control points Y	$\mathbf{p}_{S,0-3}^I$	16
Total Size			**152**

Table 1: Size, in bytes, of the data sent to the GPU per muscle, per frame.

$C_T(t)$ control points are one-dimensional and can be stored in 2 vectors. $C_S(t)$ three-dimensional control

points need 6 vectors. By using the linear precision property (see equation (4)) we can save memory space by encoding $\mathbf{p}^O_{S,0-3}$ and $\mathbf{p}^I_{S,0-3}$ points in the $2D$ basis $\mathscr{B} = \{\overrightarrow{\mathbf{S}}; \overrightarrow{\mathbf{S_d}}\}$, the missing dimension is implicitly defined by keeping a constant distance between $\mathbf{p}^O_{S,0-3}$ and $\mathbf{p}^I_{S,0-3}$ as described in section 3.1. The $\mathscr{B}$ basis is very efficient as the majority of GPU calculations are done in *section space*.

Data containing the index of the muscle influencing each vertex and the associated fat offsets $\mathbf{L_{fat}}$ is sent to the GPU only once with the rest of per-vertex data.

5.2 Bézier Spline Basis Discretization

To speed-up the computation of $C(\hat{t})$ (see equation (1)), we store the basis functions of the spline on the GPU in an array of size n. We discretize the basis for n values t_i of $\hat{t}$ and we store the results in $[1 \times 4]$ vectors of the form:

$$[\mathbf{B}^3_0(\mathbf{t_i}), \mathbf{B}^3_1(\mathbf{t_i}), \mathbf{B}^3_2(\mathbf{t_i}), \mathbf{B}^3_3(\mathbf{t_i})] \tag{15}$$

In practice, we have found $n = 100$ and $t_i = 1/n$ to produce smooth deformations. Increasing the discretization only increases the buffer's size but has no influence on the computation time and doesn't visually improve the results.

During animation, the projection of each vertex $\mathbf{V_i}$ onto the action line $A(t)$ gives the value of t for this vertex. We obtain $\hat{t}$ from equation (5). The thickness at $\hat{t}$ is the result of the dot product of the closest t_i spline basis vector we have previously stored with vector $\mathbf{p}^O_{T,0-3}$ if $t <= t_c$ and $\mathbf{p}^I_{T,0-3}$ otherwise. $C_S(t)$ in $\mathscr{B}$ basis is computed in a similar way for each of the two dimensions.

Having $C_T(t)$ and $C_S(t)$ we compute $\mathbf{V'_i}$ by applying equation (12). The final skin deformation is calculated with equation (13) or (14). All these computations are performed on the GPU.

5.3 Results

All of the timings and animations presented in this section have been computed on a dual core 3.0 GHz Pentium 4 PC equipped with a GeForce 8800 GTX graphics board under Windows Vista using the DirectX 10 API.

5.3.1 Male Gymnast

We have tested our algorithm on a male gymnast walking and jumping. The model is composed of 82000 triangles. We have attached 56 muscles to the arms, legs, chest and neck of the character (see figure 9). It took about a day for a non-skilled animator to shape the muscles, attach them to the bones and choose appropriate skin influence templates. Note that the number of triangles greatly exceeds the one of standard meshes used in regular video games. This is due to the fact that higher resolution meshes are needed to correctly appreciate the muscles' influences. We obtain real-time for as many as 25 characters, which corresponds to a total of 1400 muscles animated using both isometric and isotonic contractions. We observe a loss of $20-25\%$ in the frame rate due to the use of muscles (17 fps) compared to the use of a linear blend skinning only (26 fps).

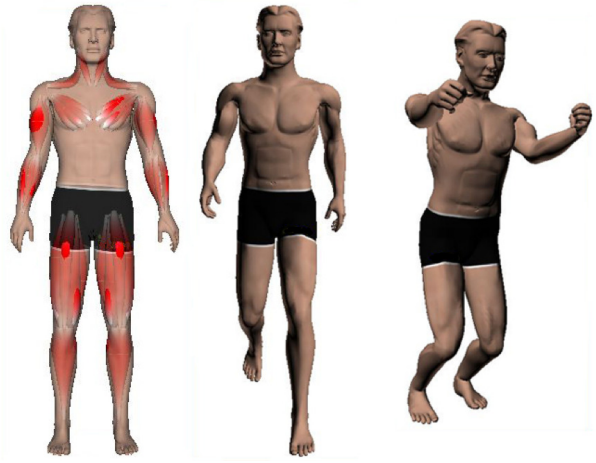


Figure 9: From left to right: *male gymnast* model composed of $82K$ polygons and 56 muscles (setup time about 8 hours); 2 frames from the GPU real-time animation.

5.3.2 Dinosaur Leg

We have also tested our algorithm on a simple dinosaur leg. The model is composed of 7800 triangles to which we have added 4 muscles (see figure 10).

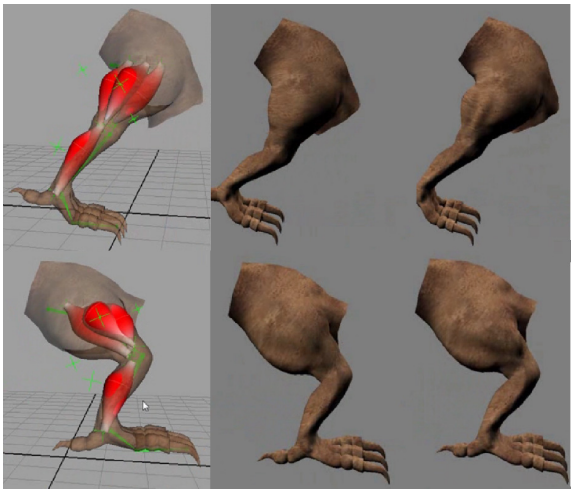


Figure 10: Two frames of the *dinosaur leg* animation. From left to right: design of 4 muscles under Maya (setup time about 20 min); Animation on GPU without muscles; Animation on GPU with muscles.

5.3.3 *Crowd Animation*

Finally, we have tested our GPU algorithm on wider crowds (see figure 11). Up to 200 instances for a total of 11200 muscles could be animated in interactive time. Note that no GPU instancing technique was used.

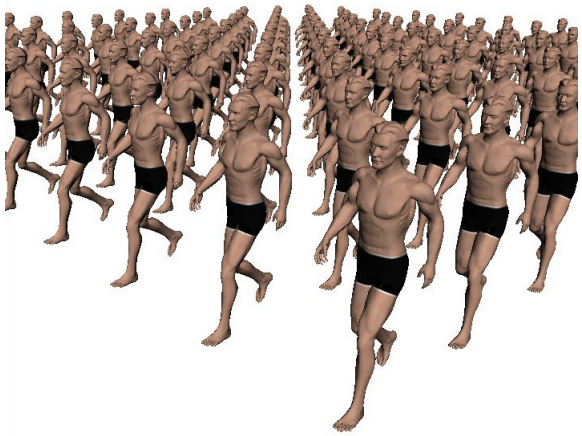


Figure 11: Crowd animation of 100 instances of the male character which corresponds to 5600 muscles total. Frame rate is 4.3 fps mainly limited by the high number of polygons: 8.2 millions.

Table 2 shows the frame rates obtained for different numbers of instances used for the man and dinosaur leg models.

Mesh	# of Instances	Total # of Muscles	Fps
Man (82*K* tri.)	20	1120	21
Man (82*K* tri.)	25	1400	17
Man (82*K* tri.)	50	2800	9
Man (82*K* tri.)	100	5600	4.3
Man (82*K* tri.)	200	14000	2.1
Leg (8*K* tri.)	100	300	32
Leg (8*K* tri.)	200	600	16
Leg (8*K* tri.)	400	1200	3.5

Table 2: Frames per second per animation for both models and a number of instances varying from 20 to 400.

6 CONCLUSION AND FUTURE WORK

We have presented a new parametric muscle model suitable for real-time character animation as well as a new skin vertices displacement algorithm as a fast alternative to physically based simulation of elasticity. Our model is suitable for both procedural animation and GPU animation. The user is provided with 2 spline curves that allow them to design a wide range of possible muscles shapes. We have thus reached our goals of accuracy, efficiency and user's usability. Compared to previous muscles models, our model is complete in the sense that all kinds of skeletal muscles can be generated (fusiform and flat); the skin layer is not rigidly attached to the muscles but slides over them without requiring the heavy cost of a physically based simulation; both isometric and isotonic contractions can be simulated thanks to our tension parameter; when deforming, the muscle preserves its volume; we offer a normal correction; our algorithm is suitable for Graphics Hardware.

Currently, there are two limitations to our work. The first one is that there is no muscle-muscle or muscle-bone interaction. While it may seem incorrect from an anatomical point of view, it may easily be overcome by the fact that the anchor points of our muscles do not need to be onto the bone. It is thus possible to keep the anchor points, hence the action line, from a certain distance to a bone or a muscle, which acts as if there were interactions. The important muscles to be modeled are the ones on the surface because they influence the outside appearance of the skin.

The second shortcoming is the modeling time. While adding muscles following an anatomy book is achievable by most users, it still takes some time to shape the muscles. One of our future work is to use medical data [Ter05a] to compute the initial shape of our muscles and provide retargetting, especially for characters of the same species.

In addition, we plan on combining our model with a model for dynamics of soft tissues whose behavior will be controlled though our *tension* parameter (when muscles are tense, the surrounding tissues jiggle less and vice-versa). We believe it should not be directly part of the muscle model because dynamics is not only due to the muscles, but also to fatty tissues. Last but not least, we would like to address the problem of automatically computing the tension of all of the muscles of a character given its animation through inverse dynamics.

7 REFERENCES

[Aub00a] A. Aubel and D. Thalmann. Realistic deformation of human body shapes. In *Proceedings of Computer Animation and Simulation 2000*, pages 125–135.

[Cap02a] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popović. Interactive skeleton-driven dynamic deformations. In *Proceedings of SIGGRAPH'02, ACM ToG*, 21(3), pages 586–593.

[Cap07a] S. Capell, M. Burkhart, B. Curless, T. Duchamp, and Z. Popović. Physically based rigging for deformable characters. *Graph. Models*, 69(1):71–87, 2007.

[Cha89a] J. E. Chadwick, D. R. Haumann, and R. E. Parent. Layered construction for deformable

animated characters. In *Proceedings of SIGGRAPH'89, ACM Computer Graphics*, 23(3), pages 243–252.

[Che92a] D. T. Chen and D. Zeltzer. Pump it up : Computer animation of a biomechanically based model of muscle using the finite element method. *Computer Graphics*, 26(2), 1992.

[Gra] Z. Gray and M. Hutchinson. Procedural muscle and skin simulation. http://www.fourthdoor.com/muscle/.

[Jam02a] D. L. James and D. K. Pai. Dyrt: Dynamic response textures for real time deformation simulation with graphics hardware. In *Proceedings of SIGGRAPH'02, ACM ToG*, 21(3), pages 582–585.

[Kav05a] L. Kavan and J. Zara. Spherical blend skinning: A real-time deformation of articulated models. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2005*, pages 9–16.

[Kav07a] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan. Skinning with dual quaternions. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2007*, pages 39–46.

[Kav08a] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4), 2008.

[Lar04a] C. Larboulette, M.-P. Cani. Real-Time Dynamic Wrinkles. In *Proceedings of Computer Graphics International 2004*, pages 522–525.

[Lar05a] C. Larboulette, M.-P. Cani, and B. Arnaldi. Dynamic skinning: Adding real-time dynamic effects to an existing character animation. In *Proceedings of Spring Conference on Computer Graphics 2005*, pages 87–93.

[Lee06a] S.-H. Lee and D. Terzopoulos. Heads up! biomechanical modeling and neuromuscular control of the neck. In *Proceedings of SIGGRAPH'06, ACM ToG*, 25(3), pages 1188–1198.

[Lee07a] K. S. Lee and G. Ashraf. Simplified muscle dynamics for appealing real-time skin deformation. In *Proceedings of International Conference on Computer Graphics and Virtual Reality*, 2007.

[Lee09a] S.-H. Lee, E. Sifakis, and D. Terzopoulos. Comprehensive biomechanical modeling and simulation of the upper body. *ACM Trans. Graph.*, 28(4):99:1–99:17, 2009.

[Lee12a] D. Lee, M. Glueck, A. Khan, E. Fiume, and K. Jackson. Modeling and simulation of skeletal muscle for computer graphics: A survey. *Found. Trends. Comput. Graph. Vis.*, 7(4):229–276, 2012.

[May13a] Maya. Autodesk, 2013.

[Moh03a] A. Mohr and M. Gleicher. Building efficient, accurate character skins from examples. In *Proceedings of SIGGRAPH'03, ACM ToG*, 22(3), pages 562–568.

[Ned00a] L. P. Nedel and D. Thalmann. Anatomic modeling of deformable human bodies. *The Visual Computer*, pages 306–321, 2000.

[Pra05a] M. Pratscher, P. Coleman, J. Laszlo, and K. Singh. Outside-in anatomy based character rigging. In *Proceedings of the ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2005.

[San08a] S. I. Park and J. K. Hodgins. Data-driven modeling of skin and muscle deformation. *ACM Trans. Graph.*, 27(3):1–6, 2008.

[Sch97a] F. Scheepers, R. E. Parent, W. E. Carlson, and S. F. May. Anatomy-based modeling of the human musculature. In *Proceedings of SIGGRAPH'97, ACM Computer Graphics*, pages 163–172.

[Ter03a] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw. Finite volume methods for the simulation of skeletal muscle. In *Proceedings of ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2003.

[Ter05a] J. Teran, E. Sifakis, S. S. Blemker, V. Ng-Thow-Hing, Cynthia Lau, and Ronald Fedkiw. Creating and simulating skeletal muscle from the visible human data set. *IEEE Trans. on Visualization and Comp. Graph.*, 11(3):317–328, 2005.

[Tur93a] R. Turner and D. Thalmann. The elastic surface layer model for animated character construction. *Proceedings of Computer Graphics International'93*, pages 399–412.

[Wan02a] X. C. Wang and C. Phillips. Multi-weight enveloping: least-squares approximation techniques for skin animation. In *Proceedings of SCA'02*, pages 129–138.

[Wan07a] R. Y. Wang, K. Pulli, and J. Popović. Real-time enveloping with rotational regression. *ACM Trans. Graph.*, 26(3):73, 2007.

[Web07a] O. Weber, O. Sorkine, Y. Lipman, and C. Gotsman. Context-aware skeletal shape deformation. *Computer Graphics Forum*, 26(3), 2007.

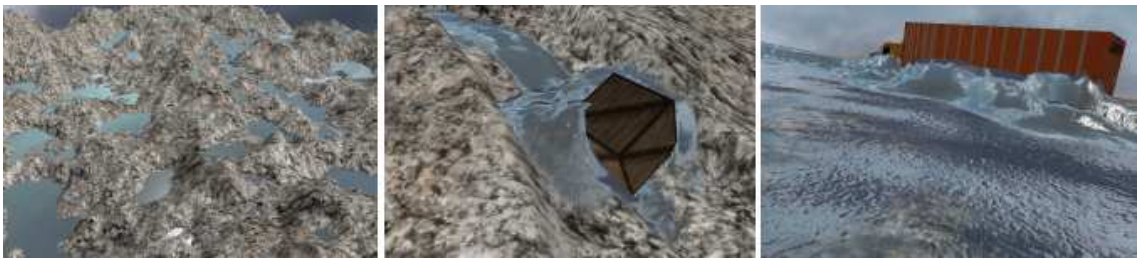
[Wil97a] J. Wilhelms. Animals with anatomy. *IEEE Computer Graphics and Applications*, 17(3):22–30, 1997.

[Wil97b] J. Wilhelms and A. Van Gelder. Anatomically based modeling. In *Proceedings of SIGGRAPH'97, ACM Computer graphics*, pages 173–180.

[Zhu98a] Q. H. Zhu, Y. Chen, and A. Kaufman. Real-time biomechanically-based muscle volume deformation using fem. *Computer Graphics Forum*, 17(3):275–284, 1998.

Interaction with Dynamic Large Bodies in Efficient, Real-Time Water Simulation

Timo Kellomäki
Tampere University of Technology
Department of Pervasive Computing
P.O.Box 553
FI-33101 Tampere
timo.kellomaki@tut.fi



Left: Natural lakes form on uneven terrain. Center, right: Examples of bodies blocking flow.

ABSTRACT

Water is an important part of nature. Interactively simulating large areas of flowing water would be a welcome addition to many virtual worlds, but the simulation is computationally demanding. Another problem is combining the simulation with rigid bodies, which are the most common interaction solution in virtual worlds. Heightfield water simulation is fast, but is especially hard to couple with rigid bodies: Usually water simply flows through the bodies. We propose a method that generalizes the extremely fast virtual pipe method to handle large, dynamic bodies. Our method diverts water around the objects. This enables us, for example, to dynamically build and destroy dams on rivers in a large virtual world.

Keywords

computer animation, games, rigid bodies, virtual worlds, water simulation

1 INTRODUCTION

Water is a fascinating element because it exhibits such complex behaviour. It is also always present in many forms in our daily lives. In video games and other real-time virtual worlds, rivers, lakes, and oceans are often implemented as static geometry with a variety of procedural techniques to improve the visual quality. As hardware improves, the trend is towards large and interactive worlds, where everything is allowed to change. The traditional approach is not always versatile enough in these situations. In many applications, the water flow will need to be simulated to see how streams behave and where lakes form.

There are several approaches to water simulation. Full 3D engineering methods can be extremely accurate and can capture the complex behavior of water down to the smallest detail. However, they are also extremely taxing on computational resources and often very complex. Unfortunately, they rarely run in real-time unless the simulation resolution is very small.

Luckily, virtual environments often only need to be convincing instead of realistic. Many of the details can be filled in using visual tricks. On the other hand, real-time performance is needed, and often even more, because the limited computational resources need to be split between several competing subsystems. Besides just the near-photorealistic graphics with a plethora of effects, also artificial intelligence and other such tasks need their share. If a water simulation system is to become popular in these applications, it needs to run in real-time with only a few percent of the available resources and be extremely easy to adopt.

There are many ways to simplify the traditional methods. One of the most common is to resort to heightfields, which reduce the number of cells needed from $O(n^3)$ to $O(n^2)$, where n is the number of cells per dimension (which determines the resolution). This means that even if hardware and algorithms progress to a point where full 3D large-scale simulation is viable in real time, heightfield methods will be able to simulate even larger areas (or with better resolution, whichever

is needed). Heightfield methods are unable to represent many interesting water phenomena, such as waterfalls or splashes. However, some of the shortcomings can be compensated by adding a particle system [OH95, HW04, CM10].

Our motivation is that, in our experience, even the simplest methods seem to be too much of a burden on resources for large-scale adoption, but the lack of realism does not pose such a problem. Therefore, we have chosen to build on the simplest possible model, based on virtual pipes.

The dynamics of virtual worlds is nowadays routinely based on off-the-shelf physics engines, which typically simulate rigid bodies. The bodies should naturally also interact with any simulated water that is present in the scene. The two simulations need to be coupled in two ways: Water affects the bodies via buoyancy, advection, and drag. On the other hand, the bodies affect the water, since water cannot enter them. This gives rise to waves and redirects the flow around the bodies.

Heightfield water simulation methods often implement an uneven terrain as another heightfield. Interaction with the terrain is well-established and easily implemented in many methods [MDH07, TMFSG07, CM10, Kel12]. However, two-way coupling with general, dynamic rigid bodies is more difficult. Water effects on objects are often easily handled, but object-to-water coupling causes problems. In all implementations we have seen, water enters the bodies and flows through them. Usually loosely physically based waves are generated around the object. This approach works when a small object is floating on a calm water surface, but makes no sense if the objects are large or the water flow is very dynamic (imagine floodwater from a dam break).

We present a novel coupling method that redirects the flow to avoid obstacles naturally. Water never enters the bodies. The method enables, e.g., building dams out of dynamic bodies. We focus on interaction with large and heavy objects that can block the flow. Therefore, we currently only implement the object to water coupling. That is, the objects do not float and are not pushed by the water. We do have plans to extend the model to cover these effects.

There is no perfect solution for the coupling, if the water is represented as a single heightfield. The objects may divide the water to arbitrarily many layers in the vertical direction and cause air pockets. We chose to simplify the situation by assuming that the bodies occupying each cell are vertically continuous and that there are no air pockets below the water surface. We ignore bodies that are above the water and do therefore not interact with it.

Our method is a generalization of the fast virtual pipe method, and runs easily in real-time for large grids using parallel computation enabled by the modern GPUs.

This paper is structured as follows. Section 2 recaps the relevant previous literature. The pipe method is described in Section 3. Our novel coupling algorithm is presented in Section 4. Section 5 evaluates the present work and establishes directions for future work.

2 RELATED WORK

In real-time animation, more or less procedural methods are often used to create waves on water. Examples of these methods are those based on the Fourier Transformation [Tes99]. These methods often create visually spectacular results for very large areas. However, they are mostly limited to visual effects on static areas of water. Dynamic flow over uneven terrain or rich interaction with moving rigid bodies does not fit naturally to these approaches.

Fluid simulation, on the other hand, has long traditions in the engineering discipline and is most often based on solving the *Navier–Stokes equations* (NSE) numerically. While this approach yields realistic results, such methods are very computation-intensive. These solvers can be divided into two categories. Eulerian methods subdivide the domain into a grid where the fluid properties are observed. Lagrangian methods track discrete particles as they are advected along the flow.

Since the full 3D methods are out of scope for this work, we refer the interested reader to Bridson [BMF07] on Eulerian methods and Solenthaler and Pajarola [SP09] on Lagrangian methods.

One hybrid method that is relevant to us is the tall-cell method, where a 3D simulation is employed near the surface, but a 2D simulation is used for deeper parts. An impressive recent example using an adaptive grid is presented in [CM11]. We especially find this method very promising for the future, since it only needs $O(n^2)$ cells just as heightfields, but still has many of the advantages of full 3D methods. However, the algorithms are very complex and the performance is still orders of magnitude slower than some of the 2D methods for a given horizontal resolution [Kel12].

2.1 Heightfield Methods

Almost all practical solutions still resort to heightfields, and the rest of this paper concentrates on those. They are naturally suited for large masses of water, such as rivers and oceans, where most of the water is relatively calm, since only the water surface is simulated.

Additional requirements for a heightfield water simulation method are dynamic free surfaces and interaction with a heightfield terrain. Free surfaces are needed if

the part of the domain that is occupied by water is allowed to change. Heightfield terrains are widely used in applications and allow for much more interesting situations than mere open-water simulations.

The *Shallow Water Equations* (SWE) are a simplification of the NSE, and can be solved to create a fast, yet realistic heightfield water simulation [LvdP02]. Recently, Chentanez and Müller added breaking waves and made many other improvements, yielding a fast method suitable for large areas [CM10].

An approach based on wave particles was proposed in [YHK07], and some researchers use Lattice Boltzmann methods [LWK03, GCTW10]. Both of these methods are well suited for open waters, but to our knowledge, have not been extended to cope with uneven terrain and dynamic free surfaces.

The simplest heightfield method is based on modeling *virtual pipes* between columns. This method was pioneered by Kass and Miller [KM90], who simplified the SWE further to reach the 2D wave equation on the water surface. They already note that the results are not realistic enough for engineering, but are promising for visualization purposes. O'Brien and Hodgins [OH95] extended the method by adding particles to overcome the limits of heightfields and used an intuitively simple virtual pipe formulation, which we adopt in this paper.

The model has then been developed further. Mould and Yang [MY97] added a heightfield terrain below the water and generalized the pipe model to consist of several layers per column, which adds realism by removing the assumption of vertical isotropy. Holmberg and Wünsche [HW04] combined the model with particles to simulate natural phenomena such as rivers and waterfalls. Maes *et al.* [MFC06] implemented the method on the GPU.

The pipe method is naturally parallel and therefore straightforward to implement efficiently on the GPU. The method is extremely fast and simple, yet captures the main dynamics of water flowing on irregular terrains. The most obvious downside is the lack of vortices, which create much of the interesting behavior of water. However, even this simple method would be a huge improvement to many current virtual worlds that completely lack dynamic water.

Even more interesting scenarios can be simulated if the heightfield terrain is allowed to be dynamic. All of these problems have been solved for the pipe method (e.g. [MDH07]) and SWE (e.g. [CM10]. For an impressive example of these techniques, see the Augmented Reality Sandbox [1]. However, since the terrain is still always a heightfield, even more dynamic situations can be achieved using generic rigid bodies.

[1] `http://idav.ucdavis.edu/~okreylos/ResDev/SARndbox/` (cited March 8, 2013)

2.2 Coupling with Rigid Bodies

There exists a large literature of sophisticated solid-fluid coupling methods for full 3D methods. See, e.g., [AIA+12] for a recent solution in the Lagrangian framework and [RMEF09] for an Eulerian approach. However, the 3D methods are still rarely real-time, so they typically aim for much more realistic solutions than is affordable in our context of large-scale real-time simulation. Therefore, we concentrate on solutions applicable to heightfield-based fluids.

The effects of water on rigid bodies are typically easy to model in all of the heightfield methods and many publications include some kind of a solution [YHK07, CM10, OH95, MY97, TMFSG07]. The body-to-water effect is more difficult, since even a single, convex body violates the heightfield assumption when submerged. It is obvious that there can be no physically based solution without generalising the heightfield somehow, e.g., to consist of more than one layer.

Previous solutions concentrate on small, floating objects or objects that are dropped into the water. Various methods are used to generate waves. In practically all methods we are aware of, water is allowed to enter the bodies. This does solve the problem of objects breaking the heightfield assumption, but is not believable unless the object is very small.

O'Brien and Hodgins [OH95] discuss the case of an object hitting the surface. A force is estimated so that the entering mass is situated nearly on the surface. This causes an external pressure to the water, causing it to flow to the neighboring columns. The model is physically based only for an object with no volume. A similar model is adopted by Mould and Yang [MY97].

Thürey *et al.* [TMFSG07] recognize the problem of the previous methods not taking volume into account. They first mention pushing the water columns down until overlaps are removed and redistributing the removed water in the neighborhood. As they state, this method would run into problems if the object is fully submerged, because the water does not close the gap above the object.

Thürey *et al.* citeThurey therefore propose a method where the volume of displaced water is tracked in each cell. The change in this volume is distributed to neighboring cells and can be negative, closing up the gaps caused by submerged objects. This corresponds closely to our virtual volume described in Section . In their method, water still enters the volume of the body and flows through it.

To our knowledge, our approach is the first heightfield-based method where water does not enter the bodies. We see this as a compulsory first step towards more physically based coupling with rigid bodies.

3 THE PIPE MODEL

Our version of the pipe model is mostly the same as in Mei *et al*. [MDH07], but the model is briefly repeated here to help the reader. We have made only minor changes and additions.

The scene is divided to a uniform grid of square cells. The variables that are tracked at each cell center (x,y) are terrain height $h(x,y)$ and depth of water $d(x,y)$. We denote the total water level by $H(x,y) = h(x,y) + d(x,y)$. We will often omit the cell coordinates whenever there is no danger of confusion.

Each cell is connected to its four direct neighbors (the von Neumann neighborhood) by a virtual pipe. Some versions use the Moore neighborhood, but since we are aiming for an extremely lightweight method, we do not find doubling the calculations worth the added quality.

The pipes store current outflow $f_o(x,y,i)$, where $i \in \{L,R,U,D\}$ is the direction from the cell (x,y). The outflows from a cell form the vector $f_o = [f_o(L), f_o(R), f_o(U), f_o(D)]$.

To update from time t to $t+1$, any water from external sources is first added to the depth. Then, the new flow is created by the pressure difference at the heads of each pipe. The created flow is also proportional to the cross-section of the pipe, but neither O'Brien and Hodgins [OH95] or Mei *et al* [MDH07] explicitly mention how they calculate it: Is it just an assumed constant or is the geometry of the situation taken into account?

Our solution, inspired by the cross-sections used in [MY97], is to calculate the height of the interval $I(x,y,i)$ where the water column at (x,y) touches air in the direction i, e.g., $|I(x,y,L)| = \max(h(x,y), H(x-1,y)) - H(x,y)$. The result is proportional to the pressure difference, but takes into account the case where the water level at the target cell is lower than the terrain at the source. In addition, we use a constant pipe area A that can be set by the user to change water behavior. Now, new flow is created in proportion to the height of the water–air interval in all four directions:

$$f_o^{t+1}(x,y,i) := \max(0, \mu f_o^t(x,y,i) + \Delta t \cdot A \frac{g|I(x,y,i)|}{\ell}), \quad (1)$$

where $i \in \{L,R,U,D\}$ is the direction, μ is an artificial friction coefficient as in [MY97], Δt is the time step, A is the pipe area (constant for all pipes), g is the acceleration due to gravity, and ℓ is the pipe length. $|I(x,y,i)|$ is the interval height as discussed above.

To prevent negative water levels from occurring, a factor K is calculated in each cell. It is chosen so that scaling by it limits the total outflow to the amount of water in the cell, as in [MDH07]:

$$K = \min(1, \frac{d}{\Delta t \sum f_o}), \quad (2)$$

where the sum is over the four elements of f_o and thus equals the total outflow.

Finally, the water depth is updated. For this, an inflow vector $f_i(x,y)$ analogous to the outflow vector is gathered from the scaled outflows at the corresponding neighbors. For example, $f_i(x,y,L) = K(x-1,y) \cdot f_o(x-1,y,R)$. Now, the final change of depth is calculated simply as follows:

$$\Delta d = \frac{\Delta t \cdot (\sum f_i - K \cdot \sum f_o)}{\ell^2}. \quad (3)$$

The horizontal velocity field $v(x,y)$ is needed for visual effects (and possibly water-to-object coupling in the future) and is calculated almost as in [MDH07]. For the left-right direction

$$v_x = \frac{f_i(L) - f_o(L) + f_o(R) - f_i(R)}{2\ell d}. \quad (4)$$

The calculation is similar for the up-down direction.

We handle the domain boundaries by not running the simulation on the border cells. The border cell depths are initialized to zero and never updated. This is not a realistic solution, but in the pipe method this is enough to make the borders drain water with little reflection. Other conditions could be used if needed, but this simple solution suits our purposes.

To achieve stability, a smaller timestep is required as ℓ is decreased [MDH07]. With some parameter balancing, we did not encounter any stability problems with timesteps up to $\Delta t = 25$ ms and $\ell = 1$ m. See, e.g., [MDH07] for further discussion.

4 COUPLING WITH RIGID BODIES

We now present our novel coupling method. It is a generalization of the pipe method described above in the sense that if no bodies are present, everything reduces to the basic method.

We aim for a situation where the bodies block flow, which creates most of the coupling physically. Therefore, the basic invariant in our model is that water can never exist inside a rigid body (to the precision of the simulation grid). We take that this is a necessary assumption for rich and believable water–body interaction, but it has been neglected in the previous height-field methods.

Our method assumes that each cell is blocked by a single contiguous vertical interval of bodies, $[b_-, b_+]$, where b_- is the lower limit of the interval and b_+ the upper. It is also useful to ignore bodies that are completely above the water. We assume that $b_- = \infty$ and $b_+ = -\infty$ if no body is present in a cell.

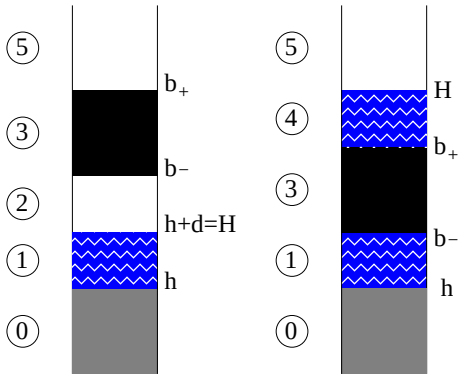


Figure 1: The layers in a single cell. Left: no water above the object (object is floating if ② is empty). Right: water above the object. In this case, water level H is calculated as terrain level + amount of water + height of body.

The method could be extended to have a number of layers similarly to what Mould and Yang [MY97] use to get rid of vertical velocity isotropy. However, the performance cost would probably outweigh the benefits for most applications. Storing several layers would make it necessary to store and update multiple depth and flow values per cell or pipe even in areas without bodies. Also drawing the result would become more complex. We therefore stay with a single-layer model.

There are two fundamentally different situations, which are differentiated by whether there is water above the blocking body. To simplify the terminology, we call these ”floating” (no water on the body) and ”submerged”, even if the floating state also includes situations where the body is not touching the water or has just hit the surface but is sinking rapidly (which is always the case in the current version of the method, since there is no buoyancy yet).

The layers in a single cell are presented in Fig. 1, where the contents are divided into six disjoint and possibly empty intervals in the vertical direction. The intervals are from bottom up: ⓪ terrain, ① lower water layer, ② air below body, ③ body, ④ upper water layer and ⑤ air above the body. Terrain continues to $-\infty$ and air to ∞. If no body is present, intervals ③–⑤ are empty (so of the two air layers, ② is chosen to exist). $|\cdot|$ denotes the height of an interval.

To reduce the storage and processing needs, we make another assumption. If there is water above a body, there is no air below it. The body in each cell is always either completely above the water, touching the water, or submerged. This means that either ② or ④ must be empty. This way we only need to store a single value for the amount of water per cell, and a single flow value per pipe. More importantly, blocking the water from entering the body becomes easier. Another benefit is that drawing the water remains simple, since a single heightfield is easy to build. On the other hand, our method erroneously blocks flow in situations such as b) in Fig. 2.

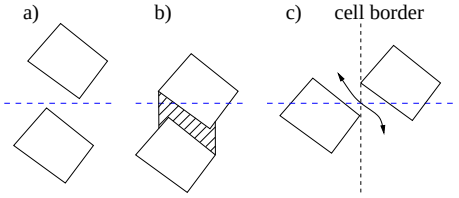


Figure 2: Some situations with multiple bodies. In a), the upper body is ignored since it does not touch water. In b), the area between bodies is erroneously blocked because of our assumption of contiguous blockers in the vertical direction. In c), two bodies are in neighboring cells and water should flow between them, but another assumption blocks this (see 4.3 for discussion).

We modify the depths described previously so that d describes the total amount of water in the cell. Some of it might be below the bodies and some above ($d = |①| + |④|$). We generalize H to mean the water surface level, taking possible bodies in the cell into account:

$$H = h + d + \begin{cases} 0 & \text{if } h + d \leq b_-, \\ b_+ - b_- & \text{otherwise.} \end{cases} \quad (5)$$

It is essential to understand that we only store h and d directly. The water level H is calculated when needed. If a body is floating, $H = h + d$, but for submerged bodies the two differ. This is demonstrated by the dashed line in Fig. 3.

For each time step, our method consists of the following phases: (a) simulate rigid bodies, (b) calculate blocked interval, (c) fix invariant and conserve volume, (d) calculate flow, (e) update depth, (f) prepare heightfield, and finally, (g) draw. The phases are described in detail below.

Phases b–f are implemented as OpenGL fragment shader passes, so each cell is processed in parallel. The method is crafted to make the cells independent and to only use information from the previous phases.

4.1 Finding Blocked Intervals (Phase b)

To find the blocked interval $b = [b_-, b_+]$ at each cell, the rigid body geometry is rendered from above and below into a texture using orthographic projection. The texture has the size of our simulation domain. Thus each texel corresponds to a single cell. The first pass writes the upper limits b_+ to a single channel of the texture using the depth buffer. The second pass inverts the depth test and writes the lower limits b_- to a second channel. The bodies that are completely above water level are not drawn.

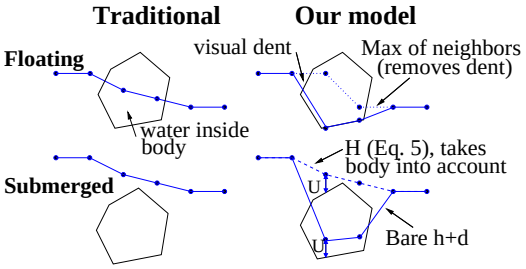


Figure 3: Comparing methods in different situations. Left: traditional solution, where water penetrates the body. Right: our method. Above: floating body, below: submerged body. Dots represent values in the cell centers. The dashed line shows calculated H, which is different from $h+d$ for submerged bodies (U is the height of water above the body). Dotted line shows the visual treatment for dents near floating bodies (see Subsection 4.4 and Fig. 7).

4.2 Invariant Update and Volume Conservation (Phase c)

After rigid bodies have moved, water may be located inside the bodies, which breaks our invariant. We need to fix this while conserving volume. This proves more difficult than one would imagine, since the blocked vertical interval can vary greatly due to aliasing caused by the grid as the body rotates. A straightforward idea would be to simply hold the amount of water above the object constant, but we found this solution unworkable because of the instability it causes. Instead, we solve this phase in a novel way that evades the instability. We first remove the old body and then add the new one. For this, we need the blocked interval at previous and current timesteps, b^t and b^{t+1}.

In this phase, we often need to add or remove water. To hold volume constant, each cell tracks *virtual volume* V: positive if water was removed and needs to be reinserted in the vicinity; negative if water was added and needs to be removed. A constant fraction τ of this volume is spread at the beginning of this step to each of the four neighbouring cells. Some volume loss could be caused if negative virtual volume ends up in areas where no water exists.

First the old body is removed. If b^t is non-empty and $b^t_+ < h+d$, the body in this cell was submerged and the volume underwater was $U_1 = b^t_+ - b^t_-$. In this case we add U_1 to the depth, holding the water surface H constant. Volume is conserved in the long run by subtracting U_1 from V, possibly causing negative virtual volume. If the body was floating, $U_1 = 0$ and nothing needs to be done.

We then insert the new body, removing any water from the interval it occupies. If $b^{t+1}_- < h+d$, the new body is underwater and the submerged volume $U_2 = \min(h+d-b^{t+1}_-, |b^{t+1}|)$. U_2 is removed from depth d and correspondingly added to V, again conserving volume. If the new body is floating, this pushes H down to b_-. Otherwise, the body is above the water, $U_2 = 0$ and H stays constant.

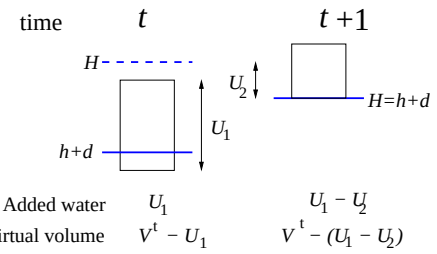


Figure 4: An example of enforcing the invariant. The submerged body at time t is removed and water amounting to U_1 is added so that H stays unchanged. In this case, the body at time $t+1$ is floating (original H is inside it), so H needs to be pushed down by U_2. The volume changes are subtracted from the original virtual volume V^t to conserve water volume.

Finally, if there is currently no body in the cell, a portion $\alpha \in (0,1]$ of the virtual volume is converted to actual water. If the virtual water is negative, water is removed, but naturally only until there is no water in the cell.

An example of removing and adding a body in this step is seen in Fig. 4. In this case, $U_1 > U_2$ so more water is added than removed. This causes some negative virtual volume, which will spread to the nearby cells and destroy a corresponding amount of water.

Virtual volume is inspired by Thürey *et al.* [TMFSG07]. They use a functionally very similar idea to remove some water from inside the bodies and propagate it to the neighboring cells, which is the core of their object-to-water coupling. However, we found that in both their and our method, the constant α needs to be rather small in order to keep the long time steps stable (we use $\alpha = 0.01$ for $\delta t = 20$ms). This means that the waves caused by the objects are also small and something more is needed.

4.3 Calculating Blocked Flow (Phase d)

Flow is governed by Eq. 1. We need to block two parts that are represented by the two terms on the right side of the equation: the existing flow (μf^t_o) and the new flow.

Let us first tackle the new flow, which is proportional to $|I|$, the height of the air–water interval. Consider the outflow from cell i to its neighbor j. The outflow can be caused by the water either above or below the object in cell i, i.e., intervals ①$_i$ or ④$_i$. The air part could be either of the two air intervals of cell j, i.e., ②$_j$ or ⑤$_j$. The intersections of these intervals are what cause new flow. The total length of water–air interface from i to j is therefore

$$|I'| = |①_i \cap (②_j \cup ⑤_j)| + |④_i \cap (②_j \cup ⑤_j)|. \quad (6)$$

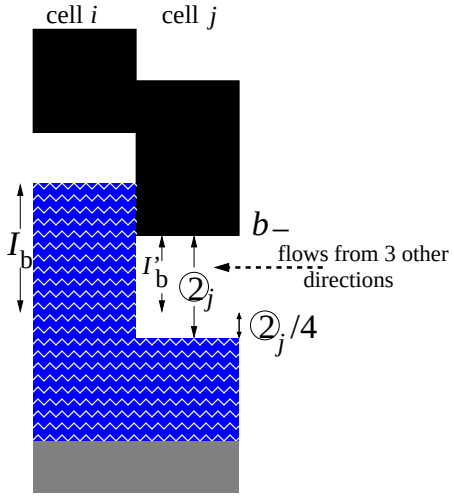


Figure 5: Limiting the existing flow below a body. I_b is the unlimited interval that is about to flow out during a time step. Limiting it by the lower border of the body in the neighbor, b_-, results in I'_b. But the binding limit here is caused by requiring that $②_j$ is not overfilled ($|I'_b| \leq |②_j|/4$, since flows come from four directions).

Recall that if no bodies are present, intervals ④ and ⑤ are empty and the formula reduces to that of the basic method.

The result of Eq. 6 is used in Eq. 1 instead of $|I|$ to calculate the new outflow vector f_o. This takes care of blocking the new flow.

Let us now handle the existing flow. We find the interval I from where water is leaving. The amount of water in the cell is now d^t and would be $d^{t+1} = d^t - \Delta t \sum f_o$ after the outflow. From d^t and d^{t+1} we calculate I using Eq. 5 with both values d^t and d^{t+1}. I is divided to parts leaving from above and below the body, I_a and I_b. If a body is present, one of them is almost always zero. If no body is present, we set $I_a = I_b = I$.

First of all, we only take into account flows from above to above and from below to below. In practice, the other flows are very rare: They can only happen when two distinct objects are right next to each other as in case c) of Fig. 2. A small change in positions would reduce this to case b) where no flow occurs anyway. Therefore this extra assumption is rarely relevant.

Now, I_a is limited from below by b_+ in the target neighbor and I_b similarly from above by b_-. The existing flows are limited in proportion to the intervals (i.e., if half of the interval is blocked, only half of the unblocked flow is allowed). Note that if no body is present in the neighbor, $b_+ = -\infty$ and $b_- = \infty$, and the flow is not limited in that case.

However, there is a complication. The interval $②_j$ at the target cell must not be overfilled, because that would cause water to enter the body. But we cannot know the inflows from the other directions without breaking the parallel structure or adding an additional iterative phase to the process. We overcome this problem with the following approximate solution.

Sum of the inflows below any body at the target cell j must be limited to $|②_j|$. Since there are four neighbours, we set a maximum of $|②_j/4|$ to each I_b going to cell j and limit the flow accordingly. This prevents $②_j$ from overfilling, but it is still filled eventually. This solution could cause artifacts in some situations, e.g., when water is flowing fast just below a bridge. However, in our experience it is an essential addition, since the lack of this limitation is easily visible, but the problems caused by it are not. The process of limiting I_b is visualized in Fig. 5.

The limited outflow from the cell is now $\max(I_a, I_b)/\Delta t$, i.e., we take the larger of the two intervals and convert it back to flow. This works both when no body is present and when either I_a or I_b is zero. The flow is further limited by K according to Equation 2, just as in the basic simulation.

A somewhat problematic situation that is unaddressed by this method happens when water flows from above a body but there is room for it below the body in the neighbor j, i.e., $|I_a| > 0$ and $|②_j| > 0$. In this case, water is transferred through the body from above to below. However, we have not noticed this visually in our tests.

4.4 Updating Depth and Constructing the Heightfield (Phases e and f)

Now that the flows have been limited, depth can be updated as in the basic method using Eq. 3. The velocity field is also calculated just as before using Eq. 4.

Before drawing the water, we need an extra phase to build the heightfield. The simplest implementation is to calculate H from the stored variables h, d, b_-, and b_+ using Eq. 5, which is indicated by the dashed line in Fig. 3. This approach causes visual dents near floating objects, because the object pushes water down below itself and thus the water mesh seems to curve down already outside the object.

We treat the denting problem by finding the cells where the object has probably pushed water down (b_- is within some ε of H). We adjust the values at these cells to be the maximum of H values in their Moore neighborhood. This value will most probably be the level from outside the body and takes care of most of the visual problems. This is demonstrated by the dotted line in Fig. 3.

Some problems still remain, however. Objects that are nearly submerged sometimes cause flicker, because small changes in d can cause large variations in H. The water-object borders are not always as clean as they could be.

Figure 6: A dam is built on a river.

Luckily, many of these problems could be masked by adding foam and particle effects to places where rapid change is happening, since the problems occur exactly where the foam would normally appear. This is an important task for future work.

5 EVALUATION AND FUTURE WORK

We have implemented the proposed method using OpenGL and GLSL shaders. The implementation uses relatively basic rendering techniques. One of the most important visual feature is visualizing the flow by using two bump textures with alternating weights. The velocity field generated by the method is used to advect these textures. The textures are reset when their weight is zero to prevent excess stretching. This method is used in many current games, e.g., [Vla10].

The rigid bodies are simulated using the open source Bullet physics engine[2]. The engine handles body dynamics and collisions. Currently the physics simulation is done on the CPU. The bounding boxes of the bodies are needed when eliminating bodies that are above water level in Section 4.3. We read back the H values and do the elimination on the CPU. This is a major performance problem that can be avoided in the future by simulating the bodies on the GPU and sharing data between the two simulations.

Our method prevents water from entering and flowing through the bodies. Water also visually seems to flow around the objects due to the calculated velocity field.

[2] `http://bulletphysics.com/`

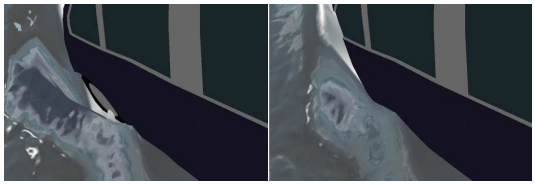


Figure 7: An example of our fix for visual dents near objects. Left: unfixed heighfield curves down already outside the object. Right: fixed border.

In addition, our method enables building dams out of dynamic rigid bodies. None of these come naturally with the previous methods that allow water to enter the bodies.

For comparison, we have implemented the rigid body coupling method suggested in Thürey *et al*. [TMFSG07]. Their method is best suited for floating objects, but also supports submerged bodies and is the state of the art in heightfield water physics coupling as far as we know. The accompanying video contains a scene where a wave reaches a large and heavy dynamic object, either passing through it almost unaffected (Thürey *et al*.) or actually going around it (our method). The video also demonstrates how our method creates a correct velocity field, which flows around the body instead of into it.

Our simulation takes approximately 18 ms on a grid of 1024×1024 on our test laptop with NVIDIA Quadro 1000M. Since our time step is 25 ms, about 1.4M cells can be simulated in real-time. More relevant results for applications are 4.7 ms on a 512×512 grid (5.3 times real-time) and 1.6 ms on a 256×256 grid (16 times real-time). Performance is currently limited by texture bandwidth, because several of the phases need a lot of information about their neighbors.

In a recent study, Kellomäki [Kel12] compares the performance of some methods. Although these performance comparisons are very rough, we can conclude that our generalized pipe method is a few times slower than the original pipe method, but still roughly an order of magnitude faster than the other methods surveyed.

The performance of our implementation greatly suffers from the fact that we have currently implemented only a single version of the simulation shaders, which are relatively complex. An obvious optimization would be to first simulate the whole scene using the simple and fast basic method, and then recalculate the comparatively small areas that are covered by the bounding boxes of rigid bodies, using the complex shaders. This would probably reduce the time used tremendously at the cost of a few extra draw calls and pipeline state changes. On the other hand, our current implementation is not much slowed by additional bodies.

One of the largest problems in our current implementation is caused by the fact that the water surface does

Figure 8: A box is dropped into a calm lake.

not just pass through the objects anymore. This causes visual problems like dents around the bodies and flickering, some of which we have already addressed, but some of which still remain. The visual problems are apparent in the accompanying video, especially when compared to the clean borders achieved by the traditional method. However, we believe further work can solve these problems.

Traditional methods for calculating water-to-body coupling cannot be used with our solution. For example, buoyancy is traditionally proportional to the difference of water level and the lower limit of the body. In our method that information is not available, because water is pushed down by the object. Further work is needed here, e.g., to enable combining the physically based approach of O'Brien and Hodgins [OH95] with our method.

We also made some restrictive assumptions and sometimes resorted to ad hoc solutions to keep performance and stability good. Mainly, the bodies overlapping each cell need to be contiguous. Violating the assumptions becomes a problem less often than one might imagine, since most practical situations only have a single body per cell, and many objects are convex in the vertical direction. The assumptions need not be fully obeyed to achieve visually acceptable results.

One of the relevant problematic cases is a small hole in a dam. This could conceivably be overcome by treating bodies that are touching the terrain in each cell as temporary parts of the terrain itself. This should be contrasted to the previous methods, which only work correctly for objects with no volume and where no dams could be built at all (except from the terrain itself). Another limitation is that no air pockets are allowed under submerged bodies, but those would typically not be visible to the user anyway.

One of the possible next steps is to learn from the more sophisticated methods used in the 3D water simulation literature. Those methods need to be adapted to work in the heightfield context and also severely simplified, since they are still much too slow for application in actual real-time virtual environments. Another possibility is using the layered model of Mould and Yang [MY97] to separate the water above and below the objects.

Several other improvements are certainly possible. Most obviously, many additional visual techniques could be added to mask the simple simulation model used. We are currently not using any particle system to create splashes and other phenomena that cannot be represented by heightfields. We are also investigating the possibility of seamlessly converting all water to particles near the bodies and using full 3D simulation in those areas.

6 CONCLUSION

This work has described a novel method for handling object-to-water coupling in the context of heightfield water simulation. Although we elected to use the pipe method, the often-used shallow water simulations could also benefit from an approach where no flow is allowed through bodies.

We have built a prototype implementation that allows completely new kind of interaction in heightfield water simulation, such as dynamically building and breaking dams using rigid bodies. The result is very fast and, despite several assumptions, works acceptably in many practical situations.

We think that even an ad hoc method is better than the current situation, where there is simply no coupling solution at all available for large objects in large-scale heightfield water simulations. Furthermore, there is no fundamental reason why further work could not extend our flow blocking based method to also handle water-to-object coupling. On the other hand, all other existing water-to-object coupling methods are inherently not physically based because of their approach where water is allowed inside the bodies.

The problem field is becoming important since many, if not most, virtual environments rely heavily on rigid body physics for interaction and dynamics. Water that passes right through large dynamic objects is simply not believable enough in this context.

Furthermore, hardware has finally improved far enough to make the fastest water simulation methods viable in actual use cases, instead of being limited to mere research prototypes. We are on our way to having interactive water as just another easily added component in video games and other virtual environments.

7 REFERENCES

[AIA+12] Nadir Akinci, Markus Ihmsen, Gizem Akinci, Barbara Solenthaler, and Matthias Teschner, *Versatile rigid-fluid coupling for incompressible SPH*, ACM Transactions on Graphics (TOG) **31** (2012), no. 4, 62.

[BMF07] Robert Bridson and Matthias Müller-Fischer, *Fluid simulation: SIGGRAPH 2007 course notes*, ACM SIGGRAPH 2007 courses (New York, NY, USA), SIGGRAPH '07, ACM, 2007, pp. 1–81.

[CM10] Nuttapong Chentanez and Matthias Müller, *Real-time simulation of large bodies of water with small scale details*, Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (Aire-la-Ville, Switzerland), SCA '10, Eurographics Association, 2010, pp. 197–206.

[CM11] Nuttapong Chentanez and Matthias Müller, *Real-time Eulerian water simulation using a restricted tall cell grid*, ACM SIGGRAPH 2011 papers (New York, NY, USA), SIGGRAPH '11, ACM, 2011, pp. 82:1–82:10.

[GCTW10] Robert Geist, Christopher Corsi, Jerry Tessendorf, and James Westall, *Lattice-boltzmann water waves*, Advances in Visual Computing (2010), 74–85.

[HW04] Nathan Holmberg and Burkhard C Wünsche, *Efficient modeling and rendering of turbulent water over natural terrain*, Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, ACM, 2004, pp. 15–22.

[Kel12] Timo Kellomäki, *Water simulation methods for games: a comparison*, Proceeding of the 16th International Academic MindTrek Conference (New York, NY, USA), MindTrek '12, ACM, 2012, pp. 10–14.

[KM90] Michael Kass and Gavin Miller, *Rapid, stable fluid dynamics for computer graphics*, Proceedings of the 17th annual conference on Computer graphics and interactive techniques (New York, NY, USA), SIGGRAPH '90, ACM, 1990, pp. 49–57.

[LvdP02] Anita Layton and Michiel van de Panne, *A numerically efficient and stable algorithm for animating water waves*, The Visual Computer **18** (2002), no. 1, 41–53.

[LWK03] Wei Li, Xiaoming Wei, and Arie Kaufman, *Implementing lattice boltzmann computation on graphics hardware*, The Visual Computer **19** (2003), no. 7, 444–456.

[MDH07] Xing Mei, Philippe Decaudin, and Bao-Gang Hu, *Fast hydraulic erosion simulation and visualization on GPU*, Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on, Nov 2007, pp. 47–56.

[MFC06] Marcelo M. Maes, Tadahiro Fujimoto, and Norishige Chiba, *Efficient animation of water flow on irregular terrains*, Proc. of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia (New York, NY, USA), GRAPHITE '06, ACM, 2006, pp. 107–115.

[MY97] David Mould and Yee-Hong Yang, *Modeling water for computer graphics*, Computers and Graphics **21** (1997), no. 6, 801 – 814, Graphics in Electronic Printing and Publishing.

[OH95] James F. O'Brien and Jessica K. Hodgins, *Dynamic simulation of splashing fluids*, Computer Animation '95., Proceedings., Apr 1995, pp. 198–205.

[RMEF09] Avi Robinson-Mosher, R Elliot English, and Ronald Fedkiw, *Accurate tangential velocities for solid fluid coupling*, Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, ACM, 2009, pp. 227–236.

[SP09] Barbara Solenthaler and Renato Pajarola, *Predictive-corrective incompressible SPH*, ACM Trans. Graph. **28** (2009), no. 3, 40:1–40:6.

[Tes99] Jerry Tessendorf, *Simulating ocean water*, SIGGRAPH course notes, 1999.

[TMFSG07] Nils Thürey, Matthias Müller-Fischer, Simon Schirm, and Markus Gross, *Real-time breaking waves for shallow water simulations*, Proc. of 15th Pacific Conference on Computer Graphics and Applications, 2007, PG '07, 2007, pp. 39–46.

[Vla10] Alex Vlachos, *Water flow in Portal 2*, ACM SIGGRAPH 2010 courses (New York, NY, USA), SIGGRAPH '10, ACM, 2010, pp. 1–54.

[YHK07] Cem Yuksel, Donald H. House, and John Keyser, *Wave particles*, ACM Trans. Graph. **26** (2007), no. 3.

Multiple Live Video Environment Map Sampling

Tomáš Nikodým
Czech Technical University,
Faculty of Electrical Engineering
Czech Republic, Prague
nikodtom@fel.cvut.cz

Vlastimil Havran
Czech Technical University,
Faculty of Electrical Engineering
Czech Republic, Prague
havran@fel.cvut.cz

Jiří Bittner
Czech Technical University,
Faculty of Electrical Engineering
Czech Republic, Prague
bittner@fel.cvut.cz

ABSTRACT

We propose a framework that captures multiple high dynamic range environment maps and decomposes them into sets of directional light sources in real-time. The environment maps, captured and processed on stand-alone devices (e.g. Nokia N900 smartphone), are available to rendering engines via a server that provides wireless access. We compare three different importance sampling techniques in terms of the quality of sampling pattern, temporal coherence, and performance. Furthermore, we propose a novel idea of merging the directional light sources from multiple cameras by interpolation. We then discuss the pros and cons when using multiple cameras.

Keywords

Image-based lighting, environment map sampling, augmented reality

1 INTRODUCTION

The *augmented reality* (AR) allows us to merge real-world images with computer generated content. AR applications have gained increasing attention over past years. We aim to enhance the believability of rendered images in AR by novel methods for real-time consistent illumination computation. We focus on methods for live acquisition of high dynamic range (HDR) environment maps in indoor environments. We compare three methods for their decomposition into sets of directional light sources by means of importance sampling.

There are three interconnected ideas discussed in this paper. Firstly, we propose a framework for live capturing of HDR environment maps and their importance sampling. Secondly, we compare three existing importance sampling techniques in terms of the quality of sampling pattern, temporal coherence, and performance. Thirdly, we introduce a novel idea of using multiple cameras to allow for the interpolation of acquired environment maps.

For efficient illumination of the scene with the acquired environment map, we approximate them by a set of directional light sources [19] with the use of importance sampling algorithms. Most of these algorithms published in the past targeted static environment maps. When dealing with dynamic environment sequences, it is important to maintain a temporal coherence of the sampling pattern to avoid visually disturbing frame-to-frame flickering. Apart from that, if the algorithm is to be used in an interactive application, the processing must run at interactive frame rates.

One of our design goals is to provide a low-cost hardware platform. Therefore we use already available mobile smart phone on the market, Nokia N900. It has a programmable camera [1] and provides sufficient computational resources to do all the processing on-chip. Thanks to the mass production of smartphones their use is cost effective compared to custom hardware. Furthermore, modern smartphones provide a broad range of connectivity (e.g.WiFi), and have a built-in camera of reasonable resolution, frame rate and quality. These features make them a suitable piece of hardware for online environment map acquisition and processing in mobile AR settings. Processing the captured images on-chip reduces the bandwidth required to transfer the data to a computer running the rendering engine. Attaching a fish-eye lens to the build-in camera provides nearly 180-degrees field of view. The smartphone can be programmed to capture a burst of varying exposure images that are fused into a single HDR image.

The framework has been proposed to be used for illumination computation in AR applications. Existing AR systems include e.g. the *Studierstube* [20] and *Spinnstube* [24]. Ideally, the virtual objects should be merged seamlessly into the real scene (see [14] for an overview of common illumination techniques). The ability to illuminate virtual objects based on the lighting conditions of the surrounding environment adds realism to the augmented scene and enhances the user experience. For a detailed discussion of existing AR for example in virtual television studios, see [9, 3].

The paper is further structured as follows. In the next section we recall the related work. In Section 3 we describe the used importance sampling algorithms in detail. The selected algorithms are then extended in Section 4 by novel techniques that allow to merge data from multiple cameras. After describing some implementation details in Section 5 we present an evaluation of the selected algorithms and their extensions in Section 6. The last section 7 concludes the paper.

2 RELATED WORK

In this section, we review existing methods for environment map sampling and discuss their suitability for sampling of live video environment maps.

Structured importance sampling proposed by Agarwal et al. [2] combines stratified and importance sampling. The proposed importance metric takes into account both surface area and integrated illumination of a hemispherical region. The number of samples can be controlled. For dynamic sequences, temporal coherence of sampling pattern is poor [12]. The computation time (tens of seconds) is far from interactive.

The algorithm based on Lloyd's relaxation, proposed by Kollig et al. [15], yields good results for static environment maps. The number of samples can be easily controlled. The time to process single environment map is dozens of seconds, too high for real-time applications. Also, the algorithm exhibits very poor temporal coherence as even local changes can cause global changes in the sampling pattern of the entire environment map.

The hierarchical importance sampling algorithm proposed by Ostromoukhov et al. [17], based on Penrose tiling, can operate at interactive frame rates. Furthermore, the sampling pattern produced by this algorithm exhibits relatively good temporal coherence. However, it is difficult to control the number of samples as it depends on the environment map being sampled.

The median cut sampling algorithm proposed by Debevec [7] is fast enough to be used in interactive systems. However, it exhibits poor temporal coherence of sampling pattern. Localized changes in illumination affect the entire sampling pattern. Another drawback of this algorithm is that the control over the number of samples is limited.

The probability density function (PDF) based importance sampling method, proposed by Havran et al. [12], targets dynamic environment maps specifically. The sampling algorithm uses an inverse transform method for hemispheres proposed by Havran et al. [11]. It is similar to the standard inversion procedure, used for importance sampling of static environment maps by Pharr and Humphreys [18] and Burke et al. [5]. However, the inverse transform method proposed by Havran et al. [12] exhibits better continuity and uniformity, which leads to better stratification of the resulting sample positions. To handle dynamic environments, the method uses two low pass filters to improve on temporal coherence. The first filter normalizes the intensity of light sources to preserve total energy of the system. The second filter suppresses high frequency movements of light sources. Invisible light source elimination and light source clustering methods are used to improve rendering performance. The method exhibits good temporal coherence, real-time performance and the number of light sources can be chosen. However, the temporal filtering causes a temporal lag for abrupt lighting changes.

The importance sampling method for dynamic environment maps, proposed by Wan et al. [23], is based on quadrilateral subdivision of a sphere. Firstly, the sphere is mapped into 2-dimensional space using the HEALPix mapping [13, 10]. A quad tree is adaptively constructed over the quadrilaterals (referred to as *quads*). At every step, the region with highest importance is further subdivided into four quads. This process is repeated until required number of samples is reached. The number of samples can be adaptively changed. The method is fast and the results for static scenes are comparable with methods such as structured sampling [2] and Penrose-based sampling [17]. It exhibits very good temporal coherence, which makes this method well suited for dynamic scene lighting.

Spatio-temporal sampling proposed by Wan et al. [22], based on Q2-Tree sampling [23], further exploits temporal and spatial coherence of environment sequences. Their method treats an environment map sequence as a volume constructed by stacking up all the frames in the chronological order. The proposed method produces a temporally coherent sampling pattern with slightly better characteristics than the original Q2-Tree sampling algorithm. A limitation of this method is that the entire environment sequence needs to be known in advance. For this reason, the method cannot be adopted for online processing of video frames in real-time.

A mobile system using environment map illumination was presented by Son et al. [21]. It uses a custom based camera. Processing of the environment maps and rendering images runs on an iPhone. The proposed system also uses markers for position tracking.

3 ENVIRONMENT MAP SAMPLING

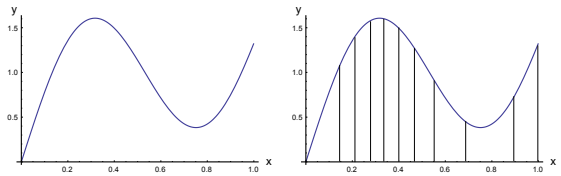
Our framework builds up on three existing environment map sampling algorithms. In this section, we provide details of the three algorithms. We then discuss our extension of these algorithms in Section 4.

Throughout this paper, we will use the following terminology. We will refer to the re-implementation of the PDF-based sampling algorithm for static environment maps [18] as *Pharr*. The extension of this algorithm for dynamic environment sequences [12], we will refer to

as *Hemigon*. The re-implementation of the algorithm based on Spherical Q2-Tree for sampling dynamic environment sequences [23], we will refer to as *Q2-Tree*.

3.1 Importance Sampling Methods based on Probability Distribution Function

Several environment map sampling algorithms based on the probability distribution function have been proposed in the past [18, 5, 12]. PDF-based importance sampling of dynamic environment maps suffers poor temporal coherence. Even if the same quaisi-random sequence is used for consecutive frames, localized changes in the environment map largely affect the global sampling pattern. As has been shown by Havran et al. [12], low pass filtering in the time domain can decrease the flickering artifacts. In this section, we recall inverse transform methods (e.g. described in [8]) based on the PDF and its application to environment maps.



(a) PDF (b) Sampled PDF

(c) CDF (d) Sampled CDF

Figure 1: Sampling of a 1-dimensional function via standard inversion method. (a) A probability distribution function (PDF), (c) corresponding cumulative distribution function (CDF). (b) and (d) are the two functions with drawn (the same) samples.

In Figure 1, we illustrate the inverse transform method on a 1-dimensional function. Firstly, a cumulative distribution function (CDF) is constructed from PDF. The construction of a CDF is computed in linear time using an iterative formulation. Then, a sequence of random numbers is drawn with uniform probability. Each of these numbers is projected via the CDF. Figure 1(d) illustrates the inversion of CDF in form $x = CDF^{-1}(y)$ that generates samples in x.

Sampling of an environment map is in principle importance sampling of a 2-dimensional discrete function, defined over hemisphere. A 2-dimensional PDF is constructed from the luminance of the pixels of the environment map. Supposing the environment map is in a latitude-longitude format, the intensity of each pixel needs to be multiplied by $sin(\theta)$ to account for smaller angular extent near the poles, where θ is the altitude angle. In practice, quasi-random number generators with uniform distribution, such as Halton sequence [8], are often used. These 2-dimensional quasi-random number vectors are then mapped via the CDFs to an altitude angle θ and an azimuth angle ϕ. This method has been successfully applied for importance sampling of static environment maps by Burke et al. [5], and Pharr and Humphreys [18].

Havran et al. [12] proposed an extension of this method to improve the temporal coherence for dynamic environment mapping. They use a mapping of PDF over hemisphere proposed by Havran et al. [11]. It avoids discontinuity for north pole and for $\phi = 0$. To decrease temporal flickering during the sampling of a dynamic environment sequence, a history of the sampling patterns for a few past frames is kept, and two low pass filters are applied, operating in the time domain.

The first low pass filter operates on the total energy of the light sources. It suppresses flickering caused by high frequency light sources, such as fluorescent tubes. The second low pass filter operates on the trajectories of the samples, suppressing high frequency movements.

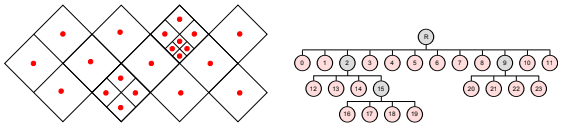
3.2 Importance Sampling Methods based on Subdivision

As opposed to PDF-based sampling methods, this class of importance sampling methods constructs a hierarchical data structure by adaptively subdividing the environment map into spherical regions [17, 23, 7, 22]. There are two approaches to the construction of such subdivisions. The first one, which we will refer to as *median split*, splits a region into two or more subregions of equal importance [7]. The second approach, referred to as *uniform split*, decides whether to further split the region or not, depending on its importance [17, 23, 22]. By region splitting two or more equally sized subregions are created. The importance metric typically takes into account the luminance of the region and its angular extent (see Equation 1). Usually one sample is placed inside each created region at the bottom of the hierarchy (leaf nodes).

Sampling patterns produced by *median split* can exhibit poor temporal coherence. This is because even local changes in the illumination affect the size or shape of subregions at the top of the hierarchy, so the sampling pattern differs dramatically. Thus, their use in dynamic environment sampling is limited. On the other hand, sampling patterns produced by *uniform split* usually exhibit good temporal coherence. This is due to the local nature of this class of subdivision methods. The size and shape of a subregion at a particular level of the hierarchy is independent of the environment map being sampled. The sampling pattern for each subregion

is not affected by illumination changes in other subregions, which stabilizes the sampling temporally.

In particular, the *Q2-Tree* sampling algorithm [23] is suitable for sampling of live video environment maps. It is fast, produces sampling patterns that exhibit strong temporal coherence and the number of samples can be adaptively changed without the impact on the temporal coherence for consecutive frames. The *Q2-Tree* algorithm is described in the rest of this section.



(a) Subdivided environment map (b) Corresponding *Q2-Tree* representation

Figure 2: Visualization of the Q2-Tree sampling algorithm.

Firstly, the sphere is mapped into 12 quadrilaterals (*quads* for short) using the HEALPix mapping [13, 10]. Then, a quad tree is iteratively constructed on the base quads. At every iteration, the quad of the highest importance is subdivided into four equally-sized subquads. All quads at the same level of subdivision have always the same surface area. The construction is complete when the required number of leaf nodes has been created. The construction is illustrated in Figure 2, showing a subdivided environment map in HEALPix mapping and a corresponding *Q2-Tree*.

The pseudo-code for *Q2-Tree* construction is given in Algorithm 1. It maintains an importance-sorted list of interior and leaf nodes. At every iteration, the left-most leaf node (highest importance) is subdivided into four quads, until the needed number of leaf nodes is created.

Algorithm 1 Sampling dynamic environment map with a *Q2-Tree*.

```
Input:
  EnvironmentMap: Spherical light probe of the environment
  RequiredNumberOfSamples: Integer

Output:
  LightSources: List of directional light sources

Algorithm:
  Q2TREE = HEALPixMappingTo12Quads( EnvironmentMap )
  EvaluateImportance( Q2TREE.Nodes )
  while(Q2TREE.NumberOfNodes < RequiredNumberOfSamples)
    NewNodes = Subdivide( Q2TREE.LeafNodes[0] )
    EvaluateImportance( NewNodes )
    Q2TREE.MoveToInterior( Q2TREE.LeafNodes[0] )
    Q2TREE.AddLeafNodes( NewNodes )
  LightSources = CreateLightSources ( Q2TREE.LeafNodes )
  return LightSources
```

During the sampling process, it is desirable to sample more densely the bright regions and less densely the dark regions. On the other hand, oversampling small bright regions should be avoided as they can be well approximated with fewer samples due to their small solid angle. The *Q2-Tree* sampling algorithm uses importance metric proposed by Agarwal et al. [2]. It combines good stratification of samples and higher density of samples in bright regions. The importance is given by the following equation.

$$p = (L)^a \cdot (\Delta\omega)^b, \tag{1}$$

where L is the total illumination of the region and $\Delta\omega$ is the solid angle. Parameters a and b are non-negative constants that are used to favor illumination (large value of a) or solid angle (large value of b) component. The result, p in the above equation, is the importance of the region. In their work, Wan et al. use constants of $a = 1$ and $b = \frac{1}{4}$ [23].

The equal solid angle property of the subdivision proposed by Wan et al. allows for computation of both terms of the importance metric in constant time. The solid angle of a quad at level i can be directly computed as $\Delta\omega = \frac{\pi}{3 \cdot 4^i}$. The illumination term is computed using a technique commonly used for texture prefiltering, known as summed area tables [6].

When the *Q2-Tree* is constructed, a single directional light source is positioned inside each of the leaf quads. The color and intensity of the light source is given by summed illumination of corresponding pixels. Wan et al. propose that the light source should be jittered deterministically around the centroid of the region to avoid regularity but maintain determinism.

For dynamic sequences of environment maps, the *Q2-Tree* need not be rebuilt from scratch for every frame. First, the importance of each region is re-evaluated, creating inconsistency in the importance-sorted list. A series of merge-and-split operations is then performed until the sorted order of the list is recovered.

4 MERGING OF SAMPLING DATA FROM MULTIPLE CAMERAS

As we have already mentioned, the acquisition and processing of the environment maps run on a smartphone. We use a wireless connection between the smartphone and the computer that runs the rendering engine (referred to as client). Such a design makes it possible to place the camera (i.e. smartphone) anywhere in the room where the lighting is to be measured. Furthermore, it is possible to place several cameras at different places. As the processing is done on-chip and the amount of data to be sent per frame is low, the number of cameras is not limited by the computational resources nor the bandwidth of the client computer. We investigated the advantages and limitations of such use of multiple cameras. We propose algorithms for merging of sampling data from multiple cameras and describe them in this section. We present the results in Section 6.

In order to compute consistent common illumination between the real and virtual objects, the illumination should be recorded at the spot where the virtual object is to be placed. This is often impractical as for example the virtual objects move in virtual television studios and it would require to move the camera together with the virtual object. Second, in AR display systems, the presence of a device inside the workspace would be disruptive for the user. It is thus inviting to approximate the illumination inside the workspace by placing several cameras around it. Furthermore, using multiple cameras could be used to improve the frame rate, temporal coherence and robustness of the system.

Before the data from multiple cameras can be merged correctly, it is necessary to perform geometric and photometric calibration. The photometric calibration requires a lux meter and is performed once for each camera. The geometric calibration needs to be repeated when the lens of the camera is repositioned. We have used external software tools based on OpenCV [4]. Multiple views of a checkerboard are taken, and the camera calibration is computed from detected corners of the checkerboard. The complete calibration of one device takes about 10 minutes to perform and does not have to be repeated unless the lens are replaced. Because of lack of space we do not discuss details in the paper.

4.1 Merging for PDF-based Methods

For the sampling algorithms based on probability distribution function (*Pharr* and *Hemigon*), the merging is straightforward. Given that the number of samples is the same for all cameras and that the samples were generated using the same sequence of quasi-random vectors, the direction, color, and intensity of each sample can be computed by linearly interpolating the direction, color, and intensity of the corresponding samples from each of the cameras.

In order for the linear interpolation of directions to be applicable, it is essential that each set of samples to be interpolated was generated from the same quasi-random vector. Typically, probability distribution function based sampling algorithms use low-discrepancy sequences, such as Halton sequence [8]. By using the same quasi-random sequence for each of the cameras, the correspondence between samples can be determined trivially from their index, supposing azimuth alignment of the cameras. Obviously, the result of such interpolation is only approximate of the real ground truth data measured in the spot, for which the interpolation was computed.

4.2 Merging for Subdivision Methods

For the *Q2-Tree* sampling algorithm, merging of data from multiple cameras is more complicated. The depth of subdivision of a particular branch can be different for each camera. Merging the Q2-Trees naively would produce non-deterministic number of samples, dependent on the environment maps. Also, the luminance of the merged tree needs to be normalized in order to preserve the total power.

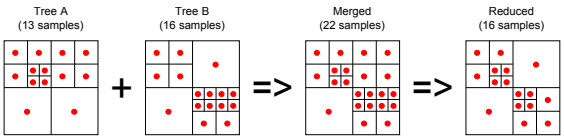


Figure 3: Visualization of the Q2-Tree merging algorithm. The two trees A and B are first merged together and later simplified to the number of samples needed.

Below, we present our *Q2-Tree* merging algorithm, given in pseudo-code in Algorithm 2. Since the original *Q2-Tree* construction algorithm uses a list-based representation of the tree, we first need to build a hierarchical structure for each of the trees to be merged. We then perform a parallel traversal of the input trees, producing a merged tree. The input trees are traversed to the maximum depth and for each node, we compute the sum of luminance of the corresponding input nodes that are available. Since the depth of a subtree can be different in each of the inputs, this step produces a tree of potentially more nodes than any of the inputs. Given camera i produced N_i samples, the merged tree can have anything between $min_{i=1}^{k}(N_i)$ samples to $\sum_{i=1}^{k} N_i$ samples. See figure 3 for illustration.

Algorithm 2 Merging of multiple Q2-Trees

```
Input:
  N: Required number of samples
  Q2[k]: List of Q2-Tree nodes for each of k cameras

Output:
  Samples[N]: List of samples
              (direction, luminance, color)

Algorithm:
  // Step 1: Tree reconstruction
  Q2Trees[] = ReconstructTrees(Q2)
  // Step 2: Merging (parallel traversal)
  MergedTree = Merge(Q2Trees)
  // Step 3: Normalization of luminance
  NormalizeLuminance(MergedTree)
  // Step 4: Importance re-evaluation (I = L^a * W^b)
  EvaluateImportance(MergedTree)
  // Step 5: Reduction to N leaves (sort-and-merge)
  FinalTree = Reduce(MergedTree, N)
  // Step 6: Sampling (Inverse HEALPix mapping)
  Samples = PlaceSamples(FinalTree)
  // Return list of samples
  return Samples[]
```

Also, the property that the luminance of an interior node equals the sum of luminances of its children no longer holds. This property is restored by the next step of our algorithm. First, the luminance of the root node is computed by averaging the luminance of the root nodes of the input trees (i.e. total luminance of the environment

map). Then, the merged tree is traversed and for each interior node, the luminance is recomputed using the following formula: $L'_i = L_i \cdot \frac{L_P}{\sum_{k=1}^{4} L_k}$, where L_i is luminance of child i, L_P is luminance of the parent, and $\sum_{k=1}^{4} L_k$ is the sum of luminances of the four children of P (including node i). Since the tree is traversed top-down and the root is already normalized, the correct value L_P is computed for each interior node before the node is visited.

We then re-evaluate the importance of each interior node, using the formula $p = (L)^a \cdot (\Delta\omega)^b$ proposed by Agarwal et al. [2]. The same values of the constants a and b are used as in the Q2-Tree construction ($a = 1$ and $b = \frac{1}{4}$). The list of interior nodes is then sorted in order of importance.

The next step reduces the number of leaf nodes to a user defined constant, using the importance-sorted list. At each iteration, we take the interior node of the lowest importance and cut off its subtree. As each interior node has four immediate children, this operation usually reduces the number of leaves by three (although a higher number is possible if the input trees differ significantly). This operation is repeated until the number of leaves drops below a user defined constant.

Now that the merged tree has approximately N leaf nodes and their luminance is correctly normalized, one sample is placed inside each leaf. The leaf nodes correspond to quadrilaterals in the HEALPix mapping [13, 10], so we use the inverse HEALPix mapping to project the sample positions back into the world coordinate system. The number of samples can be chosen with the precision of three samples, as four leaves can be always collapsed to a single leaf. The asymptotic time complexity of the proposed merging algorithm is limited by sorting of the interior nodes, $O(n \cdot \log n)$. The space complexity is linear in the number of samples.

5 IMPLEMENTATION

In this section, we present the architecture of our framework. It was one of our design goals to implement a framework that is cost-efficient, robust, extensible and easy to integrate into existing rendering engines.

The light probes (i.e. environment maps) are being captured on stand-alone devices that are capable of wireless communication. Such a device has a programmable camera and sufficient computational power to process the light probes on a chip. The on-chip processing steps include capturing a burst of varying exposure images, HDR image fusion, mapping of the captured image into the polar coordinate system, computing luminance, and importance sampling. Also, the device runs an HTTP server that provides an interface between the environment map acquisition and processing algorithm, running on a chip, and the rendering engine, running on a remote computer. Compliance with the HTTP protocol simplifies the configuration and debugging, because a standard web browser can be used to communicate with the device. The conceptual diagram is illustrated in Figure 4.

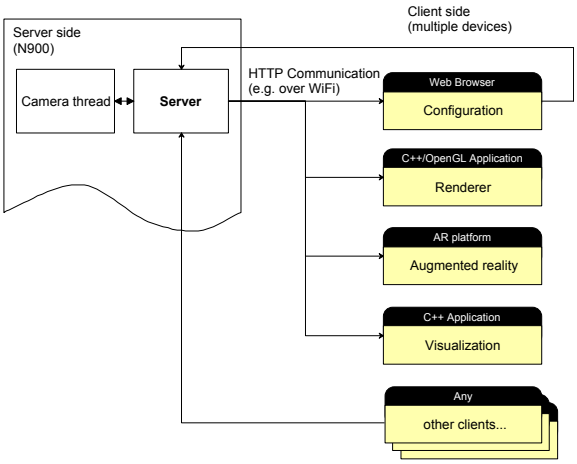


Figure 4: Conceptual diagram of our framework. The device running the sampling algorithm as a server communicates possibly with multiple clients over network.

The framework allows for multiple acquisition devices running asynchronously. The rendering engine can then communicate with all of these devices and merge the sampling data they provide. It is also possible for several client applications (rendering engines, configuration tools, etc.) to communicate simultaneously with a single acquisition device.

We have used the Nokia N900 smartphone as the environment map acquisition and processing hardware. Using a cell phone has several benefits, but also limitations. First of all, the computational resources are less powerful than those of a desktop computer. Especially the cache size is very limited. This makes it more challenging to design an efficient implementation. In particular, the benefit of accessing memory at spatially and temporally coherent locations is more pronounced. On the other hand, the small size of the device that integrates all the required features and is readily available on the market makes it a good choice for mobile augmented reality settings. It is easy to manipulate and cost-efficient.

Although we use a particular model of a cell phone (Nokia N900) in this paper, the implementation is platform independent to the extent of accessing the programmable camera and could be easily adopted to other hardware.

6 RESULTS

In this section we present the results of our work, where the test scenario is shown in Figure 5. Firstly, we

compare the three discussed importance sampling algorithms. Then, we discuss the advantages and limitations of merging the data from multiple cameras.

6.1 Comparison of Importance Sampling Algorithms

Below we summarize the results briefly for all three algorithms. Both PDF-based methods produce good results for environment sequences where the frame-to-frame changes of illumination are subtle. Temporal filtering of sample positions and intensities improves on temporal coherence but causes unwanted temporal lag if the changes are abrupt. The sampling method based on subdivision handles abrupt changes successfully, especially if the changes are local, but has problems handling subtle light source movements.

Quality of Sampling Pattern

Both implemented PDF-based importance sampling algorithms (*Pharr* and *Hemigon*) produce samples of equal power (in fact the brightness as luminance is taken as PDF). This means that the distribution of the samples is proportional solely to the distribution of the power in the captured environment map. While in general such a sampling pattern makes a good approximation of the environment map, in some cases under sampling of relatively dark regions might be a problem. Suppose we have an environment map that contains one bright light source and several much dimmer light sources. A sampling pattern that puts almost all of the samples in the small bright region would be reasonable for most views of the rendered scene. But when the main light source gets obstructed and the view being rendered is in a shadow, then the dim light sources come into play. Sampling small bright regions thoroughly and under sampling the rest of the environment produces poor results in such cases. For these reasons, it is desirable to maintain a good stratification of samples. Using such an importance metric that takes into account both brightness and angular extent of a region produces better results in these situations.

Due to the nature of HEALPix mapping, the *Q2-Tree* algorithm requires relatively many samples to approximate an environment map well (approx. 200 or more samples). The environment map is subdivided uniformly into twelve regions. An adaptive quad tree subdivision is then constructed separately on each of the twelve regions, placing one sampling into each of the leaf subregions. If we take only a few dozens of samples, the directions of light sources and thus the shadows cast in the renderer do not match the real environment and cause strong artifacts. Furthermore, for subtle movements of light sources, such as a swinging light bulb, the sampling pattern does not change, only the power of each sample changes. The rendered sequence looks unrealistic and diminishes the overall impression of the virtual environment, as the human visual system is sensitive to shadows, providing an important cue about the environment.

Figure 6 shows the sampling patterns produced by each of the three algorithms, and a scene renderer using the respective set of light sources. Note that while *Pharr* and *Hemigon* algorithms produced similar sampling pattern, the *Q2-Tree* algorithm spreads the samples across a broader area because the importance is weighted by angular extent as well as brightness.

Temporal Coherence

An important aspect of importance sampling algorithms for dynamic sequences is the temporal coherence of the sampling pattern for consecutive frames. Frame-to-frame changes of the light source positions and their intensities could cause disturbing temporal flickering in the rendered animation. The images with samples from the three algorithms are shown in Figure 7.

In the implementation of the first algorithm, *Pharr*, we used a Halton generator to generate a sequence of quasi-random vectors in two-dimensional space. Even though we use the same sequence for every frame, positions of samples change globally due to local changes. This problem is caused by the global nature of cumulative probability distribution function.

The second implemented algorithm, *Hemigon*, attempts to solve these issues by several improvements, described in Section 3.1. These processing steps increase the temporal coherence, but introduce visible temporal lag in rendered images for abrupt changes of the illumination. Despite these issues, for a typical scene with subtle frame-to-frame illumination changes, this method produces reasonable results.

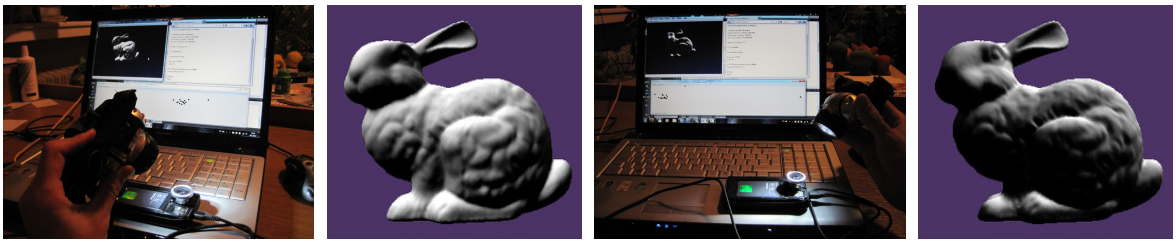
The *Q2-Tree* sampling method handles local illumination changes successfully. On the other hand, it fails to capture subtle light source movements. Furthermore, it requires more samples than the other methods to sample the environment map adequately.

We found that the PDF-based methods produce better results for scenes with subtle illumination changes and with moving light sources. The subdivision methods, on the other hand, produce better results for scenes with abrupt and localized illumination changes.

Performance

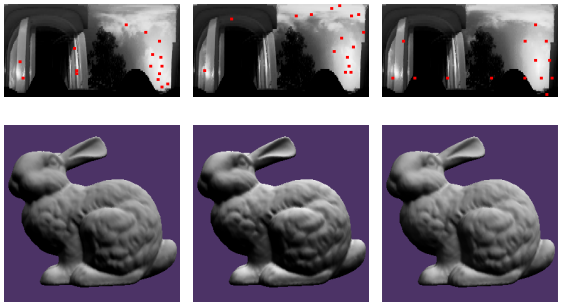
In this section, we analyze the time complexity and present the results of performance measurements of the three implemented algorithms.

The measurements were performed on a Nokia N900 smartphone. The source code was compiled in GCC compiler, version 3.4.4, with the -O3 option enabled.



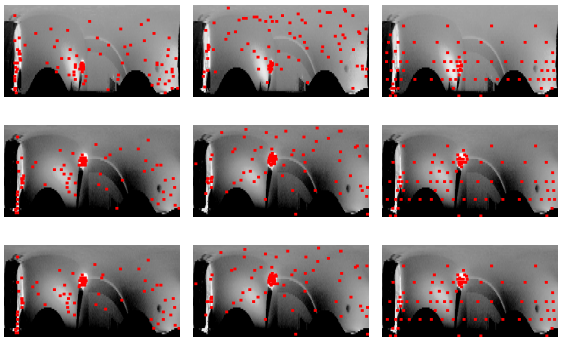
(a) Scenario 1 (photograph) (b) Scenario 1 (screenshot) (c) Scenario 2 (photograph) (d) Scenario 2 (screenshot)

Figure 5: Photographs and rendered images from a testing setup for two scenarios, when the user moves a flashlight around the camera. Changes of the illumination are interactively observed in the renderer; (a) and (b) flashlight on the left, (c) and (d) flashlight on the right.



(a) Pharr (b) Hemigon (c) Q2-Tree

Figure 6: The upper row shows the sampling patterns produced by the three algorithms for an environment map of resolution 360×90 pixels, using 16 samples; the bottom row shows a render of a bunny lit by each respective set of light sources.



Pharr *Hemigon* *Q2-Tree*

Figure 7: Snapshots of the sampling patterns produced by the three implemented sampling algorithms for three subsequent frames of a video sequence.

Three images of varying exposure were fused into one HDR image. The images were captured at resolution 640×480 pixels and mapped to a polar image of resolution one pixel per degree (i.e. 360×90 for a hemisphere). The *Q2-Tree* algorithm used a HEALPix mapping of resolution $12 \times 100 \times 100$ pixels.

The overall asymptotic time complexity is $O(w \cdot h + n \cdot (\log w + \log h))$ for the PDF-based algorithms and $O(w \cdot h + n \cdot \log n)$ for the *Q2-Tree*, where $w \times h$ is the image resolution of the environment map and n is the number of samples. The first term, $O(w \cdot h)$, accounts for the processing of the environment map prior to placement of samples. This includes, for example, the construction of the CDFs in PDF-based sampling algorithms and construction of the summed area table in the *Q2-Tree* sampling algorithm. The second term accounts for placement of samples and depends on the particular algorithm used. Post-processing of samples, including color computation, takes $O(n)$ time.

The bottleneck of the execution is the capture and processing of the HDR light probe. For a single frame, three varying exposure images are captured, fused into a single HDR image [19], and mapped into polar coordinate system. Using the configuration described above, the acquisition of a single HDR environment map takes 120 milliseconds. Table 1 shows the sampling times of the three compared algorithms, excluding the environment map acquisition time.

#samples	HDR acquisition	Sample generation		
		Pharr	Hemigon	Q2-Tree
200	120	10	150	30
1000		10	150	50
10000		80	170	430

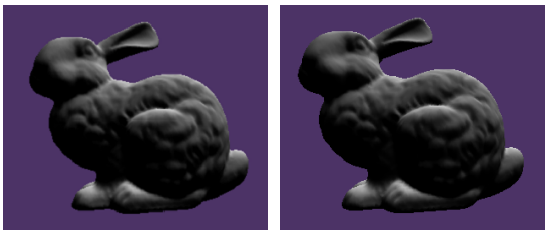
Table 1: Comparison of performance of the three discussed methods. All times are in milliseconds.

6.2 Multiple Cameras

In this section, we present the results of merging of sampling data from multiple cameras. The image renderer with light sources generated by merging the data from four displaced cameras is depicted in Figure 8 together with the reference ground truth image using samples computed from a single camera in the center. Principally the samples as a result of merging from spatially dislocated cameras can be different to the samples computed from the reference camera in dependence on the changes of indoor illumination. However, in practice we observe these differences are very small or zero. This is true only under the assumption, when

the distances between cameras and hence the environment maps are not too different.

(a)



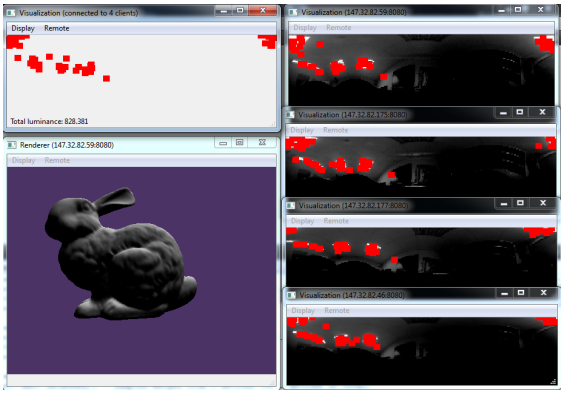
(b) (c)

Figure 8: (a) A photograph of the system in operation. The data from the four displaced cameras are merged and the results are compared against the reference camera in the center. The four outer cameras form a square sized 1000×1000mm. (b) Bunny lit by the illumination acquired on the reference camera in the center. (c) Bunny lit by the illumination computed from the four displaced cameras for *Pharr* algorithm.

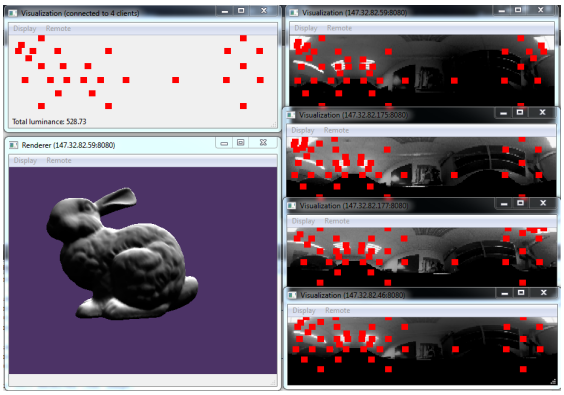
The results of merging for the algorithm *Pharr* and *Q2-tree* are shown in Figure 9. For the *Q*2-Tree sampling our merging algorithm produces the same results as averaging of the calibrated HDR light probes per pixel and building a *Q2-Tree* from scratch on the resulting image.

In order to test this hypothesis empirically, we implemented a simple testing application - it downloads the HDR light probe in .hdr format from each of the cameras, computes an average per pixel and runs the *Q2-Tree* sampling algorithm [23] on the final HDR environment map. We obtained exactly the same set of samples as produced by our *Q2-Tree* merging algorithm. Clearly, the advantage of merging *Q2-Trees* instead of averaging light probes is reduced bandwidth. The *Q2-Tree* representation of a light probe is very compact, typically not exceeding several KBytes for a reasonable number of samples, as opposed to the substantially higher memory requirements consumed by the representation of a complete HDR light probe.

In addition, the system with multiple cameras increases on robustness - it is capable of providing reasonable data to the rendering engine even if one or more of the cameras fail or has data dropout (for example, due to network problems) as long as the failure is detected. Also, the use of multiple asynchronously running cameras increases the frequency of light sources update and the data can be used to create a temporal blur, improving the temporal coherence.



(a) Pharr



(b) Q2-Tree

Figure 9: Results of merging for (a) *Pharr* and (b) *Q2-Tree* algorithm. The four windows on the right show sampling patterns and environment maps captured by four displaced cameras. The windows on the left show the resulting sampling pattern and a scene illuminated by the resulting set of directional light sources.

7 CONCLUSIONS

We presented a framework for live capturing of HDR light probes and their decomposition into sets of directional light sources. We implemented three importance sampling algorithms and compared them in terms of the quality of the sampling pattern, temporal coherence, and performance. We extended the existing techniques by merging data from multiple cameras to better approximate the lighting of the real environment, and discussed the associated advantages and limitations. We have shown a proof of concept implementation of our

ideas. Our framework with multiple cameras is useful for example in mobile setting of virtual television studios and augmented reality display systems.

As future work we want to further examine the possibilities of using multiple cameras. Existing image-based lighting methods assume only distant lighting, represented by directional light sources. This assumption is not valid, especially for indoor scenes, and causes artifacts in mixed reality applications, where the light source position is required for shadow detection and generation. By using two or more displaced cameras, it would be possible to estimate the position of point light sources and thus even improve the results of consistent illumination computation similar to the outdoor scenarios [16].

ACKNOWLEDGEMENTS

This research has been partially supported by the Czech Science Foundation under research program P202/11/1883 (ARGIE) and P202/12/2413 (OPALIS).

8 REFERENCES

[1] A. Adams, D. Jacobs, J. Dolson, et al. The Frankencamera: An Experimental Platform for Computational Photography. *ACM TOG*, 29(4):29, ACM, 2010.

[2] S. Agarwal, R. Ramamoorthi, S. Belongie, and H. Jensen. Structured importance sampling of environment maps. In *ACM TOG*, 22(3):605–612. ACM, 2003.

[3] R. Azuma. *A Survey of Augmented Reality*. Presence: Teleoperators and Virtual Environments, 6(4):355–385, August 1997.

[4] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Incorporated, 2008.

[5] D. Burke, A. Ghosh, and W. Heidrich. Bidirectional Importance Sampling for Illumination from Environment Maps. In *ACM SIGGRAPH 2004 Sketches*, pp. 112, ACM, 2004.

[6] F. Crow. Summed-Area Tables for Texture Mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, ACM, 1984.

[7] P. Debevec. A Median Cut Algorithm for Light Probe Sampling. In *ACM SIGGRAPH 2005 Posters*, pp. 33:1-33:3, ACM, 2005.

[8] G. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer, 1996.

[9] S. Gibbs, C. Arapis, C. Breiteneder, V. Lalioti, S. Mostafawy et al. Virtual Studios: An Overview. *In IEEE MultiMedia*, 5(1):18–35, 1998.

[10] K. Górski. HEALPix. *Jet Propulsion Laboratory, California Institute of Technology, NASA*, 2012. `http://healpix.jpl.nasa.gov/`.

[11] V. Havran, K. Dmitriev, and H.-P. Seidel. Goniometric Diagram Mapping for Hemisphere. In *Short Pres. (Eurogr. 2003)*, pp. 293–300, 2003.

[12] V. Havran, M. Smyk, G. Krawczyk, K. Myszkowski, and H. Seidel. Interactive System for Dynamic Scene Lighting Using Captured Video Environment Maps. In *Proc. of EGSR'05*, pp. 43–54, Konstanz, Germany, 2005.

[13] E. Hivon and B. Wandelt. Analysis Issues for Large CMB Data Sets. *Proc. Evolution of Large-Scale Structure*, arXiv:astro-ph/9812350, 1998.

[14] K. Jacobs and C. Loscos. Classification of Illumination Methods for Mixed Reality. In *Computer Graphics Forum*, 25(1):29–52, 2006.

[15] T. Kollig and A. Keller. Efficient Illumination by High Dynamic Range Images. In *Proceedings of EGWR'03*, pp. 45–50, Leuven, Belgium, 2003.

[16] Y. Liu and X. Granier. Online Tracking of Outdoor Lighting Variations for Augmented Reality with Moving Cameras. *IEEE Trans. on Visualization and Computer Graph.*, 18(4):573–580, 2012.

[17] V. Ostromoukhov, C. Donohue, and P. Jodoin. Fast Hierarchical Importance Sampling with Blue Noise Properties. In *ACM TOG (SIGGRAPH)*, 23(3):488–495. ACM, 2004.

[18] M. Pharr and G. Humphreys. Infinite Area Light Source with Importance Sampling (plugin description). On `http://www.pbrt.org`, 2004.

[19] E. Reinhard. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 2006.

[20] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. Encarnaçao, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. *Presence: Teleoperators & Virtual Environments*, 11(1):33–54, 2002.

[21] W. Son, B. Nam, T. Kim, and H. Hong. Using Environment Illumination Information in Mobile Augmented Reality. In *IEEE conference on Consumer Electronics 2012*, pp. 588–589, 2012.

[22] L. Wan, S.-K. Mak, T.-T. Wong, and C.-S. Leung. Spatiotemporal Sampling of Dynamic Environment Sequences. *IEEE Trans. on Visualization and Computer Graph.*, 17(10):1499–1509, 2011.

[23] L. Wan, T.-T. Wong, and C.-S. Leung. Spherical Q2-tree for Sampling Dynamic Environment Sequences. In *Proceedings of EGSR'05*, pp. 21–30, Konstanz, Germany, 2005.

[24] J. Wind, K. Riege, and M. Bogen. Spinnstube: A Seated Augmented Reality Display System. In *Proceedings of EGVE'2007*, pp. 17–23, 2007.

Marker-less Facial Motion Capture based on the Parts Recognition

Yasuhiro AKAGI
Kagoshima University
Korimoto, 1-21-24
Kagoshima
890-8580, JAPAN
akagi@ibe.kagosima-u.ac.jp

Ryo FURUKAWA
Hiroshima City University
3-4-1, OzukaHigashi
AsaMinami-Ku, Hiroshima
731-3194, JAPAN
ryo-f@hiroshima-cu.ac.jp

Ryusuke SAGAWA
AIST
1-1-1 Higashi
Tsukuba, Ibaraki
305-8561 JAPAN
ryusuke.sagawa@aist.go.jp

Koichi OGAWARA
Wakayama University
Sakaedani 930
Wakayama-city
640-8510, JAPAN
ogawara@sys.wakayama-u.ac.jp

Hiroshi KAWASAKI
Kagoshima University
Korimoto 1-21-24
Kagoshima
890-8580, JAPAN.
kawasaki@ibe.kagosima-u.ac.jp

ABSTRACT

A motion capture method is used to capture facial motion to create 3D animations and for recognizing facial expressions. Since the facial motion consists of non-rigid deformations of a skin, it is difficult to track a transition of a point on the face over time. Therefore, a number of methods based on markers have been proposed to solve this problem. However, since it is difficult to place the markers on a face or on an actual texture of the face, it is difficult to apply the marker-based capture methods. To overcome this problem, we propose a marker-less motion capture method for facial motions. Since the thickness of a skin varies in each facial part, the features of the motion of the each parts also vary. These features make the non-rigid tracking problem more difficult. To prevent the problem, we recognize five types of facial parts (nose, mouth, eye, cheek and obstacle) from 3D points of a face by using Random Forest algorithm. After the recognition of the facial parts, we track the motion of the each part by using a non-rigid registration algorithm based on the Gaussian Mixture Model. Since the motions of the each part are independently detected, we integrate the motions of the each part as 3D shape deformations for tracking the motions of the points on the whole face. We adopt a Free-Form Deformation technique which is based on the Radial Basis Function for the integration. This deformation method deforms 3D shapes seamlessly with pairs of key points: several numbers of points of a source face and the corresponding points of a target shape which are detected by the non-rigid registration algorithm. Finally, we represent the motion of the face as the deformation from the face of the initial frame to the others. In our results, we show that the proposed method enables us to detect the motion of the face more accurately.

Keywords

Facial animation, Motion tracking, Non-rigid registration, Deformation, Random Forest

1 INTRODUCTION

The facial animation and capturing motions are one of the important topics in the area of computer graphics and vision[1, 2, 3, 4]. The most important point for capturing motions is that the capturing method can correctly track the movement of all points (vertexes) from a frame to another frame of the motion. In case of a facial motion capture, since the movement of a skin on a face is mainly caused by a movement of muscles, it has more flexibility than that of a body movement caused by a movement of a bone and an axis. To detect the movement of a face between frames, a number of methods have been proposed. The method which tracks artificial markers on a face is the well-known approach to capture the motion of a face[3, 5]. By using the artificial markers, a capturing method can track the movement of the point robustly, even if a point on a face has fewer

features for tracking. In other cases, when it is difficult to place the artificial markers on a face or when we want to capture the texture at the same time, marker-less capturing methods are used. Some of the marker-less methods detect corresponding points between the frames based on the features of the shape[6, 7]. These feature based approaches enable us to track a movement of a mouth, nose and eyes which have strong features of its shape. However, it is difficult to track a movement of a cheek and a forehead which have a few features by the feature based approaches. In this case, the non-rigid registration algorithms[8, 9] can be a solution to track the movement of a face. If we apply the non-rigid registration algorithm for tracking, the algorithm detects the transformation from a face in one frame to that of another frame smoothly. However, since the thickness of a skin varies in each part of a face, a movement of a face also varies in each part. This feature of the non-rigid registration algorithm erases the feature of the movement of a face.

To solve this problem, we propose a motion capture method that independently tracks movements of each part of a face. For this method, we define five parts of a face: nose, mouth, eye, cheek and obstacle. To recognize the facial parts from a face, we utilize a Random Forest[10]. After the recognition of the facial parts, we detect the movement of the each facial parts with the non-rigid registration algorithm[8, 9]. Finally, we integrate the movement of each part into a single one to represent a movement of the whole shape of a face.

2 RELATED WORK

In this section, we describe the related works about the facial motion capture and the registration of 3D objects.

2.1 Facial Motion Capture

Since it is difficult to create the facial animations manually, a number of methods are proposed to capture the facial motion automatically. There are two types of approaches to capture the facial motion: a marker-based approach and a marker-less approach.

The marker-based approach has the advantage that it can more robustly detect the movement of the markers than the marker-less approaches. Huang *et al.* proposed the method to capture a high-fidelity facial movement by using one hundred markers[3]. This method can capture realistic and dynamic wrinkles and fine-scale facial details. However, the marker-based approach requires time to set up the marker and it is difficult to use to capture the scene, since the object has to be in contact with the face. Bickel *et al.* proposed the method that uses painting markers to capture the detailed deformation of the face such as the wrinkles[11][5]. These marker-based approaches have a common problem that it is laborious to put the markers and it is difficult to capture the natural texture at the same time with the motion.

On the other hand, the marker-less approaches are proposed to solve the problems of markers. Sibbing *et al.* uses the feature tracker like the KLT tracker[12] to detect the movement of a face[4]. They proposed the Surfel Fitting method to fit the deformed face to the scanned 3D points of the face. Weise *et al.* constructs the facial performance database of a person to detect the motion of the face from 2D image and 3D point set[13]. This method finds the similar poses from the database and reconstructs the facial expression by combining the similar poses. This kind of method uses 2D texture information. Therefore, the motion capture method based only on the 3D point is required and the following approaches are conducted.

2.2 Registration of 3D objects

The registration algorithm is commonly used to integrate two point sets that are scanned from different positions and angles. It is also useful to detect the movement of an object. Therefore, there are a number of approaches to capture the motion of the object by using the registration algorithms.

2.2.1 Rigid Registration

The rigid registration algorithms detect the translation and rotation of an object from two point sets. The ICP algorithm[14] is one of the well-known approaches. In this paper, we try to detect the non-rigid movement of a face. Although, the scanned data of a face is almost non-rigid, it still contains a rigid transformation. Therefore, we use the Normal Distributions Transform (NDT) algorithm[2] to estimate the rigid transformation of a face. The NDT is robustly estimating the transformation by using a mixture of a Gaussian distribution to represent the distribution of the points[15][16].

2.2.2 Non-Rigid Registration

The non-rigid registration algorithms become one of the most important topics in computer vision and graphic fields and it is useful to detect the movement of the point sets[7, 8, 6, 17]. Myronenko et al. proposed the Coherent Point Drift(CPD) algorithm[9] that regards the distribution of the point set as Gaussian Mixture Models and minimizes the distance of the Gaussian centroids. The method proposed Chui *et al.* is a similar approach that uses Expectation-Maximization (EM) algorithm with the basis function as the thin-plate spline (EM+TPS)[18]. Jian *et al.* advanced this kind of approach by using the L2 distance between Gaussian mixtures representing two point sets[8]. They compare the performance of four types of approaches: L2 distance and TPS based approach (L2+TPS), L2 distance and the Gaussian based RBF (GRBF) based approach (L2+GBRF), TPS and CPD. Since these algorithms have the acceptable robustness and accuracy, we

use the L2+TPS algorithm to detect the movement of facial parts.

Tevs *et al.* proposed a novel method which is called an Animation Cartography[7]. This method detects the correspondence of all points from a frame to the next frame by using the graph matching algorithm. This method detects the holes to interpolate them by considering the global correspondences. Li *et al.* proposed a robust reconstruction method for a moving object[6]. This method uses the template model which is made from the shape of the first frame and key points of the template. To detect the movement, they deform the template model to fit the shape of the next frame[19].

2.3 Recognition of a human body

In our approach, we recognize the facial parts from an input point data. In terms of studies on the recognition of a human body, various methods have been proposed. Shotton et al. proposed an efficient method to quickly and accurately predict 3D positions of body joints from a depth image[20]. They use the Random Forest[10] to recognize the body parts. Dantone also uses the Random Forest based approach to recognize the facial parts in real-time[21]. One of the advantages of the Random Forest is that it quickly decides the class of the input, even when the input data is huge and the feature vector is composed of large dimensions.

3 OVERVIEW

Our method is composed of four steps as shown in the Figure 1. In step (a), we create surfaces of a face from a set of point cloud generated by the 3D scanning method[22]. We use a 3D shape feature which is called First Point Feature Histogram (FPFH) [23] as a feature vector for a Random Forest to recognize the facial parts. At this time, we also create a height map (the height is equal to the depth from the camera) which is used for the step (d).

In step (b), we train the Random Forest to recognize the facial parts. Then, we classify the facial parts into five types: nose, mouth, eye, cheek and obstacle. The feature vector of each point of the face is defined by using the FPFH and a normalized position. To calculate the normalized position, we define the origin point at the tip of the nose and normalize positions of all points into -1.0 to 1.0. We create the training set by selecting the each part manually. We will describe the details of this step in Sec. 4.

In step (c), by using the results from the step (b), we detect the movement of the each facial part independently. We calculate the transformation of the each point from the point set belongs to first frame to it belongs to other frames by using the L2-TPS algorithm.

In step (d), we integrate the transformation of the each point that is estimated in the step (c) into the small number of the key points to represent the deformation of the face. We use a deformation method which is based on the Radial Basis Function (RBF)[24, 1]. Finally, the motions of the face are represented by the deformations that deform the face shape from the first frame to other frames. We will describe the details both the step (c) and (d) in Sec. 5.

4 FACIAL PARTS RECOGNITION

In this section, we describe how to recognize the facial parts from 3D points of a face (the step (b) in Figure 1). Since the number of 3D points included in the motion of a face is usually huge, it is required that the recognition algorithm has an acceptable performance to process over $100,000$ points. The Random Forest algorithm (RF) has the advantage in speed to process the huge data. On the other hand, RF has a problem called "over fitting". This is when RF has a tendency to recognize an unknown object as one category (it is difficult to reject the data) in case of a generic object recognition. In case of a facial parts recognition for motion tracking, there are a few numbers of unknown objects like outliers and obstacles. To overcome the problem, we add the obstacles into one class of the facial parts. Finally, we define the five classifications of the facial parts: "nose", "mouth", "eye", "cheek" and "obstacle". The reason why we don't differ the left and right parts is that if the number of the parts is large, each possibility value of the parts of a point is more widely spread. Then, the points which are not determined any facial parts are increased. To prevent this, we select the five classifications of the facial parts. In the following sections, we will discuss about a feature vector for recognizing the facial parts and training process of the RF.

4.1 Feature Vector for Recognizing the Facial Parts

It is important to define the feature vector for the Random Forest to recognize the facial parts exactly. A feature vector which has enough information about the 3D shape of a face is suitable for recognizing the face. Therefore, we utilize the FPFH method[23] to compose the feature vector because the FPFH efficiently represents the feature of the local shape of an object. FPFH is a feature descriptor for point clouds that consists of a histogram of the mean curvature between a point and neighboring points. Because each facial part has its own features in the curvature, we select FPFH as the feature vector. In our preliminary experiment, there are some errors when we recognize the facial parts form a face with only FPFH as a feature vector. Then, we add a normalized position of a point to the feature vector. To

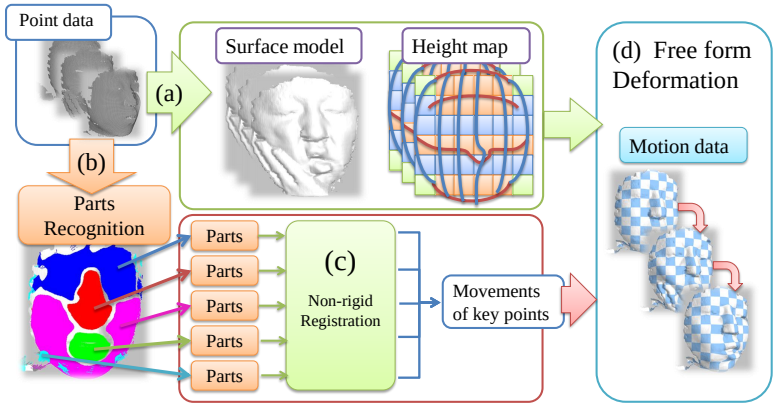


Figure 1: Overview of the marker-less facial motion capture. (a)Initial registration and triangulation form the scanned 3D points. (b)Facial parts recognition using the random forest. (c)Non-rigid registration for tracking the movement of each part. (d)Face deformations based on the movement of the parts.

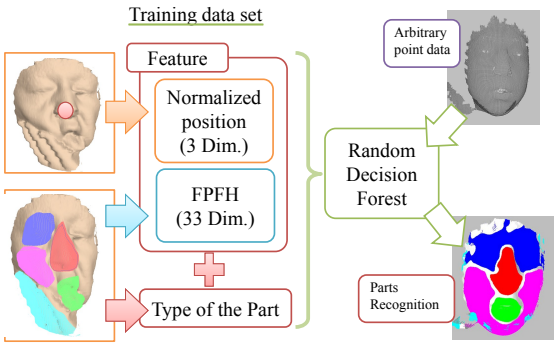


Figure 2: Flow diagram of the recognition of the facial parts.

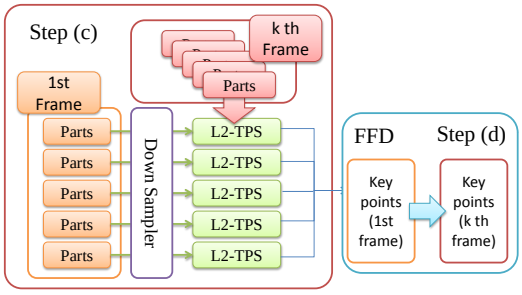


Figure 3: Flow diagram of the motion tracking and deformation of a face.

denote the normalized position, we define the center of the face as the tip of the nose and normalize the size of the face into the range of -1.0 to 1.0. This additional feature makes each decision tree of the Random Forest simpler. The feature values of the FPFH are usually similar between left eye and right eye. However those of the normalized position are different. This strong feature is useful to make the decision process of facial parts simpler. Hence, the feature vector has the 33 dimensions information from the FPFH and three dimensions from the normalized position for representing the feature of a point.

4.2 Training Process

We train the RF by using the classification of the facial parts and the feature vector. To make training sets, we choose 6 frames from the input motion and specify the area of the facial parts manually. In this approach, users have to make the training sets for every input data set. This is the limitation of our method, however this approach has the benefits that it enables us to reject scanning noises and an obstacle. Then, we also select the tip of the nose for creating the feature vector. The trained RF is used in the motion tracking process.

5 MOTION TRACKING FOR THE FACIAL PARTS

In this section, we explain about the motion tracing process of a face (step (c) in Figure 1). Figure 3 is the detailed flow diagram of the step (c) to (d) in Figure 1. We describe a set of points representing a face in t th frame as $F^t = (v_0^t, \cdots, v_i^t, \cdots, v_N^t)^{\mathrm{T}}$. And a facial part c of the face is described as $P_c^t = (v_x^t, v_y^t, \cdots)^{\mathrm{T}}$.

Before the tracking process, we recognize the facial parts P_c^t by using the Random Forest $RF(x)$. The $RF(x)$ gives the value of the possibility of each facial part for each point. We accept the point which has the possibility of over 90% as the part P_c^t(Eq. 1).

$$RD(F^t)(the\ possibility\ of\ c\ is\ over\ 90\%) \rightarrow P_c^t \quad (1)$$

In the following sections, we describe the details of the motion tracking process.

5.1 Sampling of tracking points

Our method tracks movements of each part P_c^t on the Non-rigid Registration method named "L2+TPS" (de-

scribed in section 2.2.2). Since the L2+TPS is designed to deal with less than a thousand points, it is difficult to apply our face model (it contains hundreds of thousands of points.) However, since L2+TPS is useful way to track a deformation of a non-rigid object, we reduce the number of the points for tracking (Figure 3 (a)). We apply the down sampling technique which is called approximate voxel grid filtering [25] to reduce the number of the points within 800 to 1500. We describe the reduced point set of P_c^t as $\hat{P}_c^t$. Then, since most of points of P_c^t are not tracked by L2+TPS by the reduction of points, we will calculate movements of the reduced points by interpolating the movement of $\hat{P}_c^t$ in section 6.

5.2 Discussion about the target frame of the tracking

When we track the movement of a face from a multi-frame data, there are two types of approaches for tracking the movement. One is the approach which tracks the movement between the neighboring frames (from P_c^t to $P_c^{(t+1)}$). The other is that it tracks the movement from the initial frame to other ones (from P_c^0 to P_c^t). The former approach has an advantage that since the difference between P_c^t and $P_c^{(t+1)}$ is usually small, it is easy to fit P_c^t to $P_c^{(t+1)}$ by using non-rigid registration algorithms. However, if the number of frames is large, errors that are caused by each tracking step are accumulated in large amounts. The latter approach has an advantage that even if a tracking result from P_c^0 to P_c^t contains error, the result does not affect the other tracking results. And if the shapes of a face around the end of the motion are similar to the shape of the initial frame, the tracking results will be better than the former approach. In case of facial motions, since differences from the initial shape to the other ones are smaller than those of body motions, L2+TPS can directly track the movement form P_c^0 to P_c^t. Therefor we adopt the latter approach for tracking.

5.3 Motion Tracking based on the Parts Recognition

Equation 2 represents the tracking procedural from $\hat{P}_c^0$ to $\hat{P}_c^t$ by using L2+TPS.

$$L2+TPS_{\hat{P}_c^0 \to \hat{P}_c^t}(\hat{P}_c^0) = Q_c^t \tag{2}$$

Where Q_c^t represents registered positions from $\hat{P}_c^0$ to fit to $\hat{P}_c^t$. Q_c^t and $\hat{P}_c^1$ has same number of points and the each point of Q_c^t is related to the each point of $\hat{P}_c^t$. In the following chapter, we will describe an integration process of tracking results Q_c^t to detect the movements of all points of a face F^0.

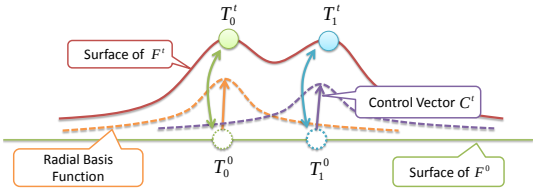


Figure 4: RBF based deformation.

6 INTEGRATION OF TRACKING RESULTS OF THE FACIAL PARTS

The tracking results Q_c^t are the sparse information about movements of each facial part c. We calculate movements from F^0 to other frames by applying a shape deformation method. In this process, since we don't consider the difference of the facial parts, we describe a tracking result of the entire face as $Q_c^t = \cup_c Q_c^t$. Now, L2+TPS registration method gives the positions of all points Q^t on the face F^t. When a shape deformation method deforms F^0 to fit to F^t, Q^0 should be able to deform to Q^t by the method. Therefore, we utilize the shape deformation method described in the following section.

6.1 Shape deformation method based on Radial Basis Function

We introduce the shape deformation method based on RBF which deforms an object by interpolating moving points[1]. The basic idea of the shape deformation is as follows (Figure 4). Now, we put a point T_0^0 on a surface $F^0 = (v_0^0, \cdots, v_i^0, \cdots, v_N^0)^{\mathrm{T}}$ and give a deformation target point T_0^t. When only the point T_0^0 moves to T_0^t, a deformed surface $\hat{F}^t$ is easy to determine by the following equation.

$$v_i^t = v_i^0 + RBF(||T_0^0 - v_i^0||)(T_0^t - T_0^0) \tag{3}$$

By using the equation 3, T_0^0 is correctly moved to T_0^t. However, if we add another pair of point T_1^0 and T_1^t, the deformed surface doesn't pass the T_0^t and T_1^t because of the movement vectors $(T_0^0 - v_i^0)$ and $(T_0^0 - v_i^0)$ affect each other by RBF. To solve this problem, the shape deformation method detects vectors which are called the "control vector" C^t. By considering the effect of moving points, the C^t is detected to make $\hat{F}^t$ passing T_0^t and T_0^t. In this study, we have the pairs of moving points Q^0 and Q^t. Then, we describe the movement of the points as $M^t = Q^t - Q^0$ and the distance between the points of Q^0 is given by the following matrix D.

$$D(Q^1, Q^1) = \begin{bmatrix} ||Q_0^1 - Q_0^1|| & \cdots & ||Q_0^1 - Q_m^1|| \\ \vdots & \ddots & \vdots \\ ||Q_m^1 - Q_0^1|| & \cdots & ||Q_m^1 - Q_m^1|| \end{bmatrix}. \tag{4}$$

Table 1: Size of frames and points of input data.

Data set	Slap	Smile	Stretch	Grip
No. of frames	46	55	48	48
No. of points	115169	190599	321142	161517

Then, the relationship of the control vector C^t and the movement of the points M^t is as following equation.

$$T^t = RBF(D(Q^1, Q^1))C^t \tag{5}$$

Since M^t and Q^1 are the known values, we can detect C^t by using the invert matrix of D (equation 6).

$$RBF(D(Q^1, Q^1))^{-1}T^t = C^t \tag{6}$$

Finally, by using the control vector C^t, an arbitrary point v_i^0 of the face F^0 is deformed to the $\hat{F}^t$ by the following equation.

$$v_i^t = RBF(D(v_i^0, Q^1))C^t \tag{7}$$

Through this process, the movement from F^0 to $\hat{F}^t$ is detected.

6.2 kernel function

We discuss a kernel function of RBF which defines the feature of the interpolation between points. Since we track the movement of the facial parts independently, it has a feature that there are a few points of Q^t between the boundaries of the parts. If we use a Gaussian like function as a kernel function of RBF, the movement of the area where the density of the points is sparse is smaller than the other area. Thus making an unnatural movement of the face. Therefore, we select a kernel function as Euclid distance $RBF(x) = ||x||$ which affects the movement to the wider area. This kernel function is also used in the research of facial deformation by Noh *et al.* [24].

7 RESULTS AND DISCUSSIONS

In this chapter, we show tracking results of four types of motions and discuss the accuracy of the results. The four types of motions are as follows: slapping a face (Slap), make a smile (Smile), vertically shrunk face turn to expand (Stretch) and it is an example of general object that a hand grips two rubber balls (figure 5). Table 1 shows numbers of points consists of the initial frame and frames of each motion.

7.1 Accuracy of the captured motion

For evaluating the accuracy of the proposed method and previous Non-rigid registration methods (L2+TPS[8], L2+GRBF[8], EM+TPS[18], CPD[9]), we calculate the mean distance between the tracked face $\hat{F}^t$ and the input faces F^t as errors of tracking. Table 2 shows the conversion of the mean distances of each method.

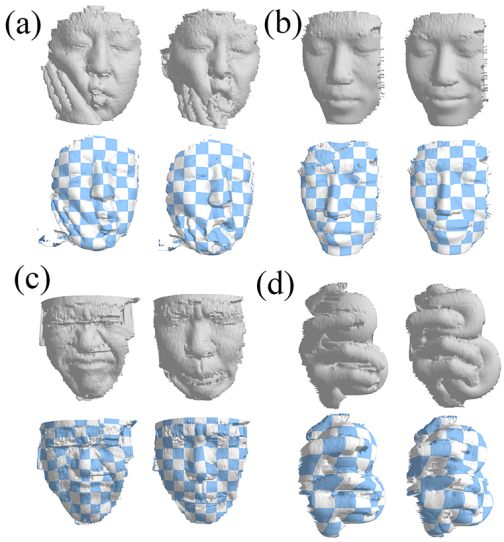


Figure 5: Examples of the input motions and tracked ones (upper row:input, bottom row:tracked). (a)Slap. (b)Smile (c)Stretch (d)Grip

Table 2: Conversion of the mean distances between the input point clouds and the captured faces of the each algorithm.

	Mean distances (mm)				
Data set	Our	L2+TPS	L2+GBRF	EM+TPS	CPD
Slap	1.70	2.23	2.37	2.38	2.35
Smile	0.56	0.78	0.54	0.55	0.79
Stretch	1.27	1.55	1.46	1.55	1.63
Grip	0.84	1.55	1.52	1.85	1.55

This result shows that the proposed method method succeeds to reduce the errors of the motion "Slap", "Stretch" and "Grip". Especially in the motion "Slap", even though this motion includes a hand contacted to the face, we can reduce the errors. Figure 6 visualizes the errors of some frames of the motion. In this result, the errors on the nose and the cheek are reduced. This result shows that by dividing the facial parts for tracing, we can reduce making wrong correspondences of the tracking.

7.2 Evaluation of identity between the initial face and the tracked face

In the section 7.1, we evaluated the distances between the initial face and the tracked one. Although it has a meaning to evaluate the fitness of both the faces, it is not clear that a point of the initial face correctly moves to the identical point of other ones. Then, to evaluate the identity between the initial face and other

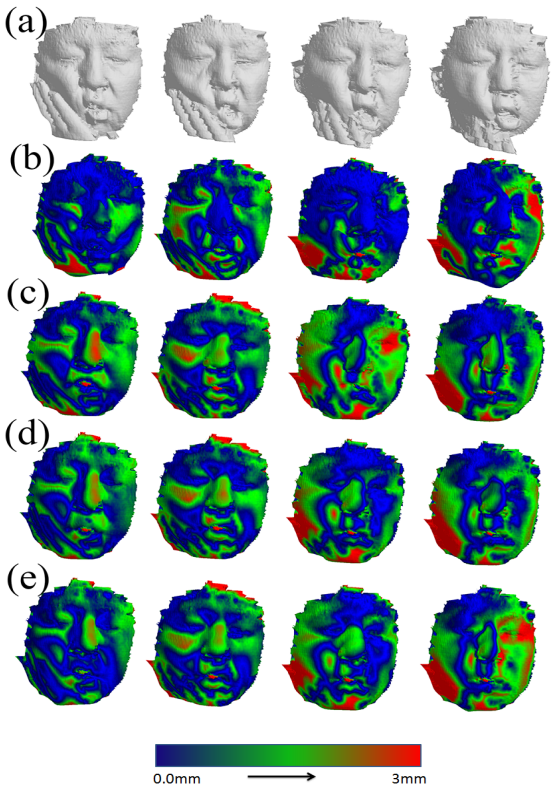


Figure 6: Visualization of errors on the face. (a)Input motion. (b)Results tracked by proposed method. (c)L2+TPS. (d)L2+GBRF. (e)EM+TPS.

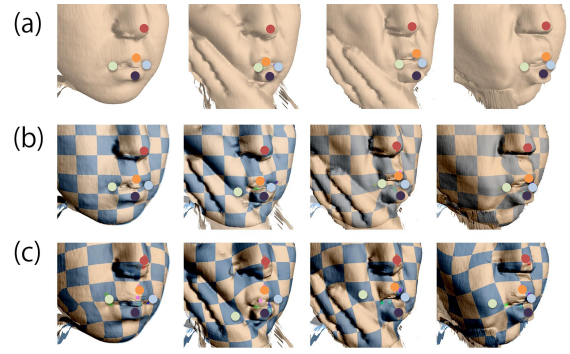


Figure 7: Accuracy of the motion. (a)Grand-truth (b)Proposed method (c)L2-TPS

ones, we manually made the ground truth of the movement around the mouth by giving same positions of five key points in each frame (the top row of the figure 7). Figure 8 shows the mean distance between the tracked points and the ground truth data. These results show that the proposed method can detect the movement of the face more accurately than other methods. However, our method has several errors in the positions of the several key points. This will be our next challenging task to improve in our method.

8 CONCLUSION

The conclusion of this paper is as follows:

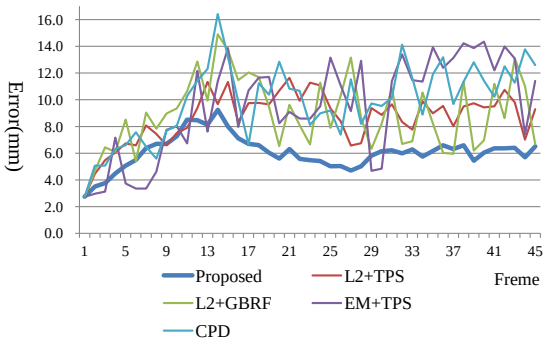


Figure 8: Conversions of the tracking error of the key points around the mouth.

- The proposed method succeeded to detect the movement of a face as the deformation from the initial shape to the other shapes.
- The recognition of the facial parts provides the efficient constraint to detect the correct transformation of the key points.
- We show that the proposed method enables us to detect the motion of the face more accurately than the other non-rigid registration algorithms.

The proposed method still has several errors to detect the shape of the facial parts. In the future, we will utilize the texture information of a face to capture more accurate motions.

9 ACKNOWLEDGMENT

This work was supported in part by the Funding Program for Next Generation World-Leading Researchers(NEXT Program) No.LR030 and Grant-in-Aid for Young Scientists(B) No.25870570 in Japan.

10 REFERENCES

[1] Botsch, M., Kobbelt, L.: Real-time shape editing using radial basis functions. Comput. Graph. Forum **24** (2005) 611–621

[2] Magnusson, M.: The Three-Dimensional Normal-Distributions Transform — an Efficient Representation for Registration, Surface Analysis, and Loop Detection. PhD thesis, Orebro University (2009) Orebro Studies in Technology 36.

[3] Huang, H., Chai, J., Tong, X., Wu, H.T.: Leveraging motion capture and 3d scanning for high-fidelity facial performance acquisition. ACM Trans. Graph. **30** (2011) 74:1–74:10

[4] Sibbing, D., Habbecke, M., Kobbelt, L.: Markerless reconstruction and synthesis of dynamic facial expressions. Comput. Vis. Image Underst. **115** (2011) 668–680

[5] Bickel, B., Lang, M., Botsch, M., Otaduy, M.A., Gross, M.: Pose-space animation and transfer of facial details. In: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. SCA '08, Eurographics Association (2008) 57–66

[6] Li, H., Adams, B., Guibas, L.J., Pauly, M.: Robust single-view geometry and motion reconstruction. ACM Trans. Graph. **28** (2009) 175:1–175:10

[7] Tevs, A., Berner, A., Wand, M., Ihrke, I., Bokeloh, M., Kerber, J., Seidel, H.P.: Animation cartography –intrinsic reconstruction of shape and motion. ACM Trans. Graph. **31** (2012) 12:1–12:15

[8] Jian, B., Vemuri, B.C.: Robust point set registration using gaussian mixture models. IEEE Trans. Pattern Anal. Mach. Intell. **33** (2011) 1633–1645

[9] Myronenko, A., Song, X.: Point set registration: Coherent point drift. IEEE Transactions on Pattern Analysis and Machine Intelligence **32** (2010) 2262–2275

[10] Breiman, L.: Random forests. Mach. Learn. **45** (2001) 5–32

[11] Bickel, B., Botsch, M., Angst, R., Matusik, W., Otaduy, M., Pfister, H., Gross, M.: Multi-scale capture of facial geometry and motion. ACM Trans. Graph. **26** (2007)

[12] Shi, J., Tomasi, C.: Good features to track. In: 1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94). (1994) 593 – 600

[13] Weise, T., Bouaziz, S., Li, H., Pauly, M.: Realtime performance-based facial animation. In: ACM SIGGRAPH 2011 papers. SIGGRAPH '11, New York, NY, USA, ACM (2011) 77:1–77:10

[14] Besl, P.J., McKay, N.D.: A method for registration of 3-d shapes. IEEE Trans. Pattern Anal. Mach. Intell. **14** (1992) 239–256

[15] Pathak, K., Birk, A., Vaškevičius, N., Poppinga, J.: Fast registration based on noisy planes with unknown correspondences for 3-d mapping. Trans. Rob. **26** (2010) 424–441

[16] Stoyanov, T., Magnusson, M., Almqvist, H., Lilienthal, A.J.: On the Accuracy of the 3D Normal Distributions Transform as a Tool for Spatial Representation. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). (2011)

[17] Zhang, W., Wang, Q., Tang, X.: Real time feature based 3-d deformable face tracking. In: Proceedings of the 10th European Conference on Computer Vision: Part II. ECCV '08, Berlin, Heidelberg, Springer-Verlag (2008) 720–732

[18] Chui, H., Rangarajan, A.: A new point matching algorithm for non-rigid registration. Comput. Vis. Image Underst. **89** (2003) 114–141

[19] Sumner, R.W., Schmid, J., Pauly, M.: Embedded deformation for shape manipulation. In: ACM SIGGRAPH 2007 papers. SIGGRAPH '07, ACM (2007)

[20] Shotton, J., Fitzgibbon, A., Cook, M., Sharp, T., Finocchio, M., Moore, R., Kipman, A., Blake, A.: Real-time human pose recognition in parts from single depth images. In: Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition. CVPR '11, Washington, DC, USA, IEEE Computer Society (2011) 1297–1304

[21] Dantone, M., Gall, J., Fanelli, G., Gool, L.V.: Real-time facial feature detection using conditional regression forests. In: Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition. (2012)

[22] Sagawa, R., Kawasaki, H., Furukawa, R., Kiyota, S.: Dense one-shot 3d reconstruction by detecting continuous regions with parallel line projection. In: Proc. 13th IEEE International Conference on Conputer Vison(ICCV 2011). (2011) 1911–1918

[23] Rusu, R.B., Blodow, N., Beetz, M.: Fast point feature histograms (fpfh) for 3d registration. In: Proceedings of the 2009 IEEE international conference on Robotics and Automation. ICRA'09, Piscataway, NJ, USA, IEEE Press (2009) 1848–1853

[24] Noh, J.y., Fidaleo, D., Neumann, U.: Animated deformations with radial basis functions. In: Proceedings of the ACM symposium on Virtual reality software and technology. VRST '00, New York, NY, USA, ACM (2000) 166–174

[25] Rusu, R.B., Cousins, S.: 3D is here: Point Cloud Library (PCL). In: IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China (2011)

Optimizing Multiple Camera Positions for the Deflectometric Measurement of Multiple Varying Targets

Oleg Lobachev Martin Schmidt Michael Guthe
Universität Bayreuth, Universitätsstraße 30, 95447 Bayreuth, Germany
{oleg.lobachev,martin.schmidt,michael.guthe}@uni-bayreuth.de

ABSTRACT

We present a device for detection of hail dents in passenger cars. For this purpose we have constructed a new multi-camera deflectometric setup for large specular objects. Deflectometric measurements have strict constraints how cameras can be placed – for instance: angular restrictions and distance limitations. An important trait of our system is the static setup – we use a *single* setup for camera configuration for *all* objects to be scanned.
We render the camera images and analyze them for the deflectometric needs to optimize the camera placement w.r.t. multiple parameters. Important ones are the positions of the cameras – reflections of the patterns should be clearly visible. Camera parameters are computed using a global optimization procedure for which we efficiently generate a good starting configuration. We introduce an empiric quality measure of a particular camera configuration and present both visual and quantitative results for the generated camera placement. This configuration was then used to build the actual device.

Keywords

deflectometry, ray tracing, quantum annealing

Figure 1: We optimize the camera positions for optimal inspection of specular surfaces using reflections of projected patterns. From left to right: an overview of the setup with color coding to show different projection screens; view through one of the cameras where the borders between screens are clearly visible; a picture of the actual device; hail damage on a (white) car where the dents are clearly visible as distortions of the reflected pattern.

1 INTRODUCTION

One of the most problematic events for car insurance companies are hailstorms as numerous claims need to be processed in the shortest possible time. Currently, each car needs to be assessed and documented by a human expert using a camera and a pattern that is reflected by the damaged surface. As this process is very time consuming, insurance companies are highly interested in automating assessment and documentation. The main challenge is reducing the recording time to process all damaged cars shortly after the event. Analysis and assessment may take longer since the claims settlement – i.e., repair – is the limiting factor anyways.

Traditional methods like laser measurement can only reconstruct an object if it consists of a diffuse material (see, e.g., [19]). For the inspection of specular surfaces, these methods are not applicable. One possible solution to this problem *deflectometry*. Cameras record reflections from a specific pattern on the surface. Once the pattern and the positions of the camera and the pattern generator (e.g., a fixed pattern or a display) are known, the surface normals can be calculated by comparing the distorted pattern reflection with the expected one.

In this joint project with i-Lumica AG, PHIcom GmbH, OGP Messtechnik GmbH, ALTRAN Group, AXA Auto Competence Center and ExactVision Bildverabeitungssysteme GmbH, we developed a specific deflectometric measurement system. The *goal* was to detect hail damage in cars, shown in the rightmost image in Figure 1. The whole measurement procedure, including driving the car in and out should take no more

than three minutes. Our setup significantly differs from the traditional approach where a human expert uses a single camera and pattern to document the hail damage:

- We consider complete *large* objects: passenger cars.
- A large *variety* of different objects – any car, not a particular brand and model – should be measured with a completely static setup.
- We do not move a single camera and pattern projector above the surface, but use a multi-camera setup with multiple projectors.

In Figure 1 the third image shows the actual measurement device. It is built using 13 back projection screens, in between cameras can be mounted on aluminum beams.

For objects with a complex form, one camera alone can not measure the complete surface. However, setting up more than one camera can mitigate this problem and lead to an almost complete coverage – we will show this fact practically. Positioning the cameras and maximizing the coverage is performed using global optimization. An important issue in our setting is the optimality of camera positioning for a *broad set* of scanned objects. We do not optimize the setup for a single particular target, but for a set of different targets.

In this paper we address the *optimization* of the camera placement. We use optimization metaheuristics to maximize the *coverage*, i.e., the amount of the surface that the cameras can inspect. Car insurance companies require at most 3% tolerance in the number of hail dents. If we reach a sufficient coverage, we can presume that the remaining area has the same density of hail dents or less. This assumption is feasible as the uncovered area is both visually and physically less accessible and thus a lower density of hail dents can be assumed. If we obtain a coverage of 94% and estimate $\sim 3.2\%$ more hail dents than found, we have 3% too many if there are none and 3% too few if the density is the same as on the covered area.

The main contributions of this paper are:

- We present and discuss our novel setup and its optimization.
- Using ray tracing we obtain qualitative measures for the *coverage* of a given camera configuration.
- We propose an algorithm for the initial camera placement that produces good starting values, essential for the subsequent optimization.
- Using quantum annealing for further optimization, we have achieved almost complete coverage ($\geq 94\%$ of the visible surface).
- We verified our method on a broad set of car models, including a car that was not used during optimization.
- We use a *static* setup, i.e. a single configuration is used for all models.

2 RELATED WORK

There are many applications for which it is necessary to reconstruct the surface of an arbitrary object. Often, only contactless measurement methods are possible [2]. Existing approaches that measure the quality and characteristics of a surface mainly depend on diffuse materials. Highly specular materials and objects with complex form are harder to measure. Two different methods form the so called *traditional techniques*. These include, but are not limited to, stereoscopic vision and triangulation.

Stereoscopic vision uses two images that are recorded from different positions to create a 3D representation of a given object [9]. The two images can be matched by a set of feature points or by dense matching. The distance to the camera can then be estimated for each pixel based on the parallax. This is also used by almost all primates for spatial vision [26]. Stereoscopic vision fails in case of specular surfaces because the points can not be matched in the exact surface plane, but only in the mirror plane, which lies behind the actual surface.

The second approach is the triangulation of incident light on a surface [19]. A laser scanline is projected onto the surface and recorded by a camera above the laser. Then the distance to the surface can be calculated from the vertical displacement using triangulation. This process depends on diffuse or partially diffuse surfaces and does not work for highly specular objects. If the diffuse reflection is too dim, the surface needs to be coated, which is not possible for all types of objects. These techniques can be used as a fallback for surfaces that are not specular enough and therefore not suitable for deflectometry.

2.1 Deflectometry

The deflectometric inspection of specular surfaces [15, 27] simulates the way how humans recognize the shape of such surfaces: we look at the distortion of the reflected patterns and infer the shape of the surface from the reflection and the original form of the pattern. A stereo setup makes it possible to infer the point cloud of the surface from a set of specular reflections [2, 20]. The reflected patterns are projected on a back light screen using a pattern generator. A camera records the reflected image of the pattern on the surface. Because the pattern has a known structure and can additionally be modified when taking several consecutive images, it clearly shows the distortions induced by the shape of the surface.

Although deflectometry has been known for 25 years, practical methods became feasible only recently [7, 14,

16, 18]. Our method differs from these in several aspects, although others also compare with a reference [2, 16].

multiple cameras Most approaches use a single camera and move it along the object. We use many fixed cameras.

static setup Our setup is not repeatedly optimized for each particular model, but the camera positions and orientations remain fixed in productive use.

multiple objects Our system scans multiple different cars without modifications of the setup in-between.

optimization We optimize the camera positions, which is not considered in most papers on deflectometry. We discuss related work for optimal camera placement below.

stereo We could also use deflectometric stereo, i.e., the reconstruction of 3D image with multiple sensors [2, 3, 18, 28]. In an interesting alternative approach, Zheng et al. [31] rotate their objects in order to spare the number of cameras and projection screens.

Directly related to our work are the papers by Balzer et al. [2] and Hong et al. [14]. The first uses a deflectometric setup to find a dent in a part of a passenger car. In contrast to our work, a single sensor and a single projection screen are moved on a robot arm. The second utilize two cameras and five screens for deflectometry of solder joints. They were not concerned with the optimization of camera positions; further we use much more cameras.

2.2 Optimization

For deflectometric measurements, the camera must be placed in such a way that the whole surface of the object is visible and entirely covered by the reflected pattern. For objects with increased complexity, such as car bodies, this is not feasible using a single camera and pattern only. Multiple cameras must record the surface. The necessary condition is that every point on the surface can be seen by at least one camera. This is known as the *art gallery problem* and has received profound research attention. While this is necessary for any measurement, it is not sufficient. The cameras must view the surfaces in such an angle that every point on the surface is also covered by a reflection of the pattern [8, 11, 23].

In mechanical measurement apparatuses, the placement of cameras can not be arbitrary. Certain conditions must be met, such as traverses and beams that can be fitted with cameras and there must be a minimum distance between two cameras. As the installation of cameras is tedious and time-consuming, a preceding optimization of camera positioning is crucial to build the device. Camera position optimization [13] was most often researched for a single camera. The random walk method [29] is often used for camera positioning, while some other approaches, like genetic algorithms [24], are viable. See also the references in Olague and Mohr [24].

Random walk. This optimization method [6, 29], abbreviated here with RW, is a typical approach to camera placement [30]. It is often used in computer vision [5, 12, 22]. The camera positions and target directions are "jittered" a bit with random values, then the new position is evaluated. If it is better than the previous one it is kept and otherwise discarded. This method often gets stuck in a local optimum if the step size is too small.

Adaptive simulated annealing. The adaptive simulated annealing method, in short ASA, is a global optimization metaheuristic [17]; it is adaptive because the step size is adjusted to the current state of the problem. ASA has an important property of being a *global* optimization method. "Usual" optimization algorithms are local, i.e., as soon as the local optimum is found, no better solution is reached. In contrast, the ASA can "move away" from the local optimum in a search for a global optimum.

Quantum annealing. This method [1, 10] (abbreviated QA) slightly differs from ASA. To achieve global maximum, ASA allows certain "setbacks": the result might be worse temporarily. In contrast, QA "tunnels" through regions between local optima, which also allows QA to find a global optimum. More formally, in QA the tunneling distance for the current step determines possible candidates for the next state; in ASA this depends on the notion of temperature in the annealing analogy. QA can be seen as a quantum Monte-Carlo method. QA is also similar to RW, the key difference lies in how the "jitter" radius is defined, depending on the step count. In RW this radius is constant. In QA the radius is chosen based on a normal distribution. The variance of this distribution is steadily decreasing, but very large, "tunneling" jumps are still possible, although less and less probable with increasing number of steps.

While ASA and QA could be used in our case, the latter has several advantages. The possible camera positions do not form a connected manifold and thus jumps are inevitable. While this is natural in QA, it needs to be specially handled in ASA. In addition, QA has been evidenced to need relatively few iterations compared to ASA in order to find a good solution [21]; in our case iterations are quite costly for setups with many cameras and objects.

3 MEASUREMENT SYSTEM

Our setup includes 13 projection screens placed in a "greenhouse" structure as shown in Figure 1. Note the spaces between the screens. First of all, this is a construction-induced limitation; however, we also use these spaces to place cameras without obstructing the projection screens. The placement of the cameras on

the supporting beams induces some limitations to the camera placement, as discussed below in Section 3.1.

Each camera records several images using different patterns on the projection screens. Based on that data, the hail dents are detected as shown in Figure 2. To register the dents on the car and remove duplicate ones, we need to determine the exact position of the car. For this purpose, we mount four wide-angle cameras: to the left, to the right, behind and above the car. Then we use background subtraction [25] to detect the car. We compare the captured images to the views of 3D models, minimizing the difference between the covered pixels. This gives us the position and actual reference model of the car. Note that the reference car models are segmented, as we need to assign the hail damage to construction parts. Projecting the images back onto the reference model gives us the exact positions of the hail dents. Subsequently, duplicates are removed. The final list per part is the result of the deflectometric measurement.

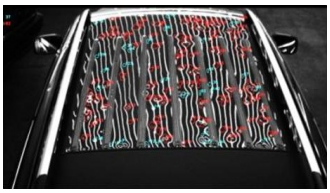


Figure 2: Camera image showing detected hail dents. In addition, the dents are classified into those that can be repaired without much effort (cyan circles) and those that cannot (red circles).

3.1 Constraints

Our approach considers a number of constraints regarding the installation and then optimizes the complete camera set. All parameters are subject to a global optimization procedure that will generate the parameter values with the highest coverage possible with respect to the specified constraints.

We change both *position* and *orientation* of the camera, as well as the focal length of the lens. Hence, each camera theoretically possesses 7 degrees of freedom: 6 DOFs for the camera position and orientation and one for the focal length, which we choose from a fixed set. The position is limited by the construction constraints: the cameras are placed on beams between the projection screens (see Figure 3) and the orientation is also limited to 2 DOFs. So in total there are only $3+1$ DOFs per camera. A further constraint is the *distance* between cameras. Due to construction limitations two cameras need to be almost 10 cm apart. We set this distance to exactly 10 cm in our optimization.

Summarizing, some constraints are implied by the physical properties of the cameras: focal distance, depth of field, resolution of the sensor. Other constraints emerge

Figure 3: Camera mounted between two projections screens. Note that while the view direction can be modified, the up vector of the camera is fixed.

from the physical construction: we place cameras not everywhere, but only where it is mechanically feasible.

We define several criteria that need to be fulfilled for each point on the surface of the car body in order to be classified as covered. These include basic visibility (visible by at least one camera), extended visibility (visible by at least two cameras if 3D reconstruction is desired), visible reflection of pattern display, and reflection adequate for deflectometry. In the following, we will discuss these criteria in detail.

Notably, surface parts with interreflections (e. g., concave parts of the car surface) are marked as not covered.

3.2 Visibility

The most basic requirement for coverage is the visibility of each point. This is easily determined: If a point can be seen by at least one camera, it is visible. To compute the visibility of a certain point, several steps are performed. The images resulting from all of the cameras in the scene are processed. Covered pixels are transformed into covered patches on the surface of the object. Any area not covered by a camera is not visible.

3.3 Reflections of pattern display

For each point, the eye vector describes the direction from which the camera sees this point. A reflection can occur when the reflection vector $\vec{R}$ belonging to the eye vector $\vec{V}$ hits one of the pattern displays. Figure 4 illustrates this concept.

One problem during the optimization with respect to the visibility of reflected patterns are the gaps between the display segments. They result from technical and mechanical constraints during construction of the system.

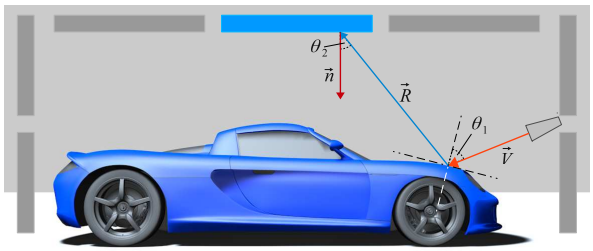


Figure 4: Fresnel reflection angles on the surface and display.

Solving this problem is easy, as the affected points have to be covered by a different camera.

The reflection quality is the most interesting and difficult requirement. An adequate reflection is one that fulfills the deflectometric requirements [16]. For our system, we can reduce these requirements to the following:

- Possible reflection
- Actually visible reflection
- Reflected image must not hide details.

The possible reflection is already guaranteed by the preceding paragraph. It is necessary for any further calculations. Whether the reflection is actually visible depends on several factors. The viewing angle influences the strength of reflections on the surface. This is known as the Fresnel effect. The angle between surface normal and viewing direction θ_1 differs between points on the surface. The glass panels of the projection screens induce another limitation of the viewing angles. The angle θ_2 between the normal of the screen surface and the view vector $\vec{R}$ must not be too large, otherwise the Fresnel reflection creates an overlay on the pattern. In our experiments, we found that up to 60° the reflection does not cause measurement artifacts.

Furthermore, we want to maintain a good quality of the reflected image. Shrinking[1] the image in one direction leads to severe loss of resolution. We do not want the shrinking in lateral direction to exceed $1:2$. This leads to the second limitation: the viewing angle θ_1 between surface normal and view vector $\vec{V}$ must be less than $\theta_{\max}$, the angle where the lateral shrinking is 0.5. Any angles larger than $\theta_{\max} = 60^\circ$ are still visible, but lead to distortions due to the extreme shrinking of the reflected pattern. Since we want to avoid such errors, $\theta_{\max}$ is the upper bound for the acceptable reflection angle.

The final restriction is the minimum resolution on the surface. The projected pixel should be smaller than a given threshold in both directions. Note that the projected size is not only distance and focus dependent, but also changes with θ_1.

4 OPTIMIZATION

The goal of our simulation approach is to optimize the camera placement in a scene of multiple cameras and an object. The more parts of the object's surface are visible, the better the placement is. Therefore the *coverage*, a measure for covered surface, is an important objective function. We aim to optimize the input parameters (position and view direction of every camera in the setup) leading to the best possible coverage. The simulation data was used to determine the number of required cameras and their optimal position and orientation for the construction of the actual physical device, see Figure 1.

The *coverage* is the value we are maximizing, i.e., the quality measure of the camera placement. We define it as the relative car body surface area, which is seen by a camera and covered with a suitable pattern reflection. We combine the individual coverages for all cameras to obtain the total covered area per model. Then we combine the relative coverages of all models using

$$x_{\max} = \arg\max_{x \in D} \left(\min_{1 \le i \le N} \left\{ \frac{A_{\mathrm{cov},i}(x)}{A_{\mathrm{total},i}} \right\} + \frac{\lambda}{N} \sum_{i=1}^{N} \frac{A_{\mathrm{cov},i}(x)}{A_{\mathrm{total},i}} \right),$$

where D is the search space, as discussed in Section 3.1, and i iterates through the N car models. We use the penalty weight λ to increase the convergence rate. In this manner not only the worst case is optimized, but also the others. We use quantum annealing to compute $x_{\max}$ and thus maximize the coverage.

4.1 Computing the coverage

We use ray-tracing on commodity graphics hardware to simulate the recorded images for a set of cameras and then compute the covered area $A_{\mathrm{cov},i}$ of each car. During this process, we estimate the total area of the car body surface in which the pattern reflection is clearly visible and which meets all constraints discussed in Section 3.1. Basically, we add up all pixels of the specular car surface for that the above constraints are satisfied. This value is then divided by the total externally visible area that is computed before starting the optimization.

Each car model is prepared in order to efficiently measure the covered area. The model is first split into reflective parts that need to be measured – i.e., the car body – and the remaining parts that may occlude the reflective ones or the screens. In addition to calculating per face and per vertex normals for the reflections, a texture map is created by unwrapping the car body using simple box mapping [4]. The Figure 5, left, shows the generated texture atlas of a car body. We use the texture map to determine if a pixel is externally visible: we simply trace several rays outward and check them for occlusion.

For each optimization pass the view of every camera is rendered to determine which parts are visible and contain usable reflections. Then each pixel of the camera view is mapped into the texture atlas. As the only information we need for this mapping is the texture coordinate of the pixel, we simply store that in the render buffer. For pixels that do not meet the constraints, we simply store invalid coordinates (e.g., $[-1,-1]$). Figure 6 shows the content of this render buffer for one of the cameras.

[1] Strictly speaking, there are two kinds of lateral shrinking of the pattern: if the surface curvature is too large and if the projected pattern is viewed under a too sharp angle. Both are important for our application, but in the following we discuss the one that is actually controllable with camera placement.

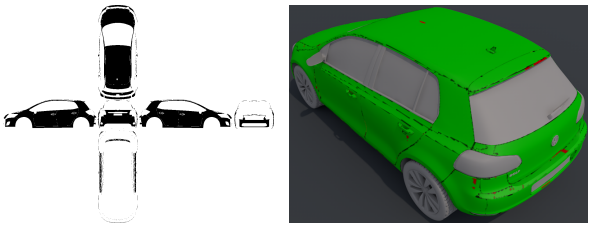


Figure 5: Texture atlas of a sample car (left) and computed coverage mapped onto the model (right). Almost everything is green, meaning sufficient coverage.

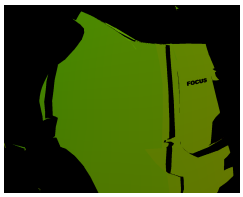


Figure 6: Intermediate buffer storing the texture coordinates of each pixel containing a suitable reflection of a pattern. These images are generated for every camera and mapped into the texture atlas.

4.2 Initial placement

All random-walk based methods, including QA, are *local* search methods. Good start values are crucially important for the quality of the result and the runtime. The first phase generates an initial placement of all cameras.

We place a camera, to cover the largest "hole". These are detected by low pass filtering the texture atlas using a Gaussian filter of radius 0.1 of the texture size. Then we search for the local maxima and keep the largest 10. To place a camera, we choose randomly one of these and determine position and normal at the closest texel containing a part of the surface. Based on position and normal, we first place the camera 1 m above that point. Then we choose a random position within a distance of 1 m from that point and project the camera to the closest free location on a beam. This process is repeated 100 times for each camera and the placement with the highest coverage is chosen. Note that the filtering and search are performed once for each camera, so the additional overhead is relatively low compared to QA iterations.

Note that the initialization search phase is not *required*, as QA would have found good solutions anyway, but greatly improves the speed of the computation.

4.3 Output

To visualize the quality of the currently found solution and build the actual device, we produce several outputs. These are a texture depicting the current coverage, a text output of the minimum coverage, and the camera placement itself. Using color-coding, we can show each point's characteristics. Pixels that are covered are green and the others are red. In addition, pixels that are not visible from above or beside the car are black. The latter is, e. g., the case for pixels covered by the license plate, in joints, or in the bottom-facing parts of the car. The right image in Figure 5 shows such a visualization.

100 cameras, 1000 iterations

Method	RW	SA	QA	IP+QA*
Time, s.	8344.34	7851.55	6995.35	3256.38

Table 1: Left columns: run time for three optimization methods, right: mean run time for the performance-tuned QA method with initial placement phase, designated "IP+QA*". All approaches use 100 cameras, we measure the time for 1000 iterations. Time is in seconds.

The textual output of the simulation is the list of cameras with their position in the construction coordinates and their look-at point. We also implemented a script to import these in 3D-modeling/CAD software to visualize the cameras and for the construction of the actual device.

5 RESULTS

We optimized the camera positions for *twelve different* car models. These models were chosen to be as variative as possible to ensure covering maximal shape space.[2] The cameras we use have a 6.9 mm $\times$ 5.5 mm CCD sensor with a resolution of 1280×1024 pixels. The cameras are capable of producing images at 100 fps. These parameters where chosen because of the sharp capture time limitations. The focal lengths of the lenses are 9 mm, 12 mm, 16 mm, 25 mm, and 35 mm. As desired resolution on the car body surface, we set a minimum of one pixel per mm. Relatively small resolution of the cameras is compensated by their speed; more cameras are added to achieve the desired resolution on the car surface. We found the minimum number of cameras required to achieve the desired coverage of 94% to be 99 for our test set. While the minimum coverage was 94.0006% (Chevrolet Lumina), the peak coverage was 97.0595% (Volkswagen Golf 6). Still remaining red spots (see Figure 7) are outliers outside the desired coverage. The high number of cameras stems from the desired accuracy of ~ 0.5 mm for the detection.

The optimization was performed on an Intel Core-i7-3770K at 3.5 GHz with 16 GB RAM and NVIDIA GeForce GTX 680, running Windows 7. For the quantum annealing we used $\lambda = 10^{-3}$. The number of iterations is 10000, disregarding heat-up. In each iteration we either changed the position of a single camera or moved all cameras. In the first case the distance of the camera "jitter" ranged from 50 cm to 0.1 mm, and in the

[2] The models were: Audi R8, BMW X5, Chevrolet Lumina, Fiat Grande Punto, Ford Focus ST, Volkswagen Golf 6, Mazda MX5, Mercedes C, Peugeot 107, Porsche Carrera GT, Renault Clio 2, Toyota Auris.

second case from 5 m to 1 mm. The cool-down factor was $\varepsilon^- \approx 0.99915$, the heat-up factor was $\varepsilon^+ \approx 1.1857$. Each time, we found a better solution, the temperature was increased by that factor. In total, the optimization took 25.5 hours. The optimization time is less than the three days required to set up the rest of the device before the cameras could be mounted.

We compared the number of required cameras for all cars against the number required when optimizing for each car separately. Figure 7 shows the coverage–number of cameras relation. For all 12 cars in a static setup, the number of cameras (99) roughly doubles the number required for each car alone (35–67).

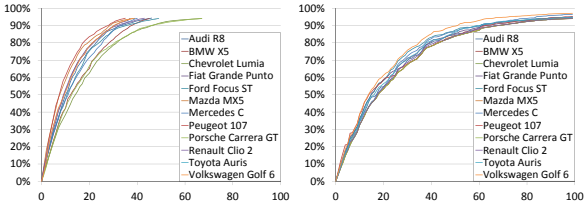


Figure 7: Comparison of coverage when optimizing for each car independently (left) vs. optimizing for all cars together (right). The horizontal axis represents the number of cameras, the vertical axis shows the coverage.

Figure 8 shows the coverage visualized on some of the car models. Most important here is the top view as hail dents are much more common on horizontal surfaces. The rightmost image shows an experiment to test our approach. After optimizing the cameras for the 12 cars, we used this setup to compute the coverage of the first generation Pontiac Firebird, which was not included in the optimization set. We reach a coverage of 93.87%. Hence, an acceptable coverage is also possible for cars that were not considered during the optimization of the camera placement. This makes it easy to use the camera setup for cars that are newly introduced to the market, because the shape space changes only marginally in short time. Still, the setup needs to be revised at some point as those chances in shape space can add up and need to be covered. The bigger the shape space is during optimization, the less revisions are due. This has also a positive effect on custom car alterations, e. g., spoilers. Surface parts occluded by these are less prone to dents.

6 CONCLUSION & FUTURE WORK

We have presented an approach to build a deflectometric measurement system for large specular objects using a multi-camera setup. The objects measurable by our setup are complete cars – a quite large field of work. The device we describe here was actually built and is now operational at AXA, Bern. We use quantum annealing combined with an effective initial camera placement to determine the optimal camera placement. We consistently reached high coverage rates for multiple simulated vehicles using a single configuration – the set of camera positions, orientations and focal lengths, shared for all vehicles. Approximately 48% more cameras are required for all cars than for the most difficult one alone.

There are some issues to investigate in our future work. Other build forms for the supportive constructions for projection panels and camera mounting could be viable. It could make sense to try some further optimization methods on the whole setup, not only for the cameras. The cameras also have further advanced parameters one could change during the optimization, like the tilt for the selective focus. We will further investigate the possibility to dynamically group the cameras for 3D reconstruction without reference models. It would also be possible to use multiple *camera triples* for a 3D reconstruction. We have tested this approach in our simulation, but do not use it in the actual device due to the higher number of cameras and availability of reference models. While 3D reconstruction is not necessary in the current system, it could be of interest for future projects.

ACKNOWLEDGMENTS

Most of this research has been conducted while the authors were with University Marburg, supported by HMWK LOEWE KMU program and PHIcom GmbH.

REFERENCES

[1] B. Apolloni, C. Carvalho, and D. de Falco. Quantum stochastic optimization. *Stoch. Proc. Appl.*, 33(2):233–244, 1989.

[2] J. Balzer, S. Höfer, and J. Beyerer. Multiview specular stereo reconstruction of large mirror surfaces. In *Proc. CVPR '11*, pages 2537–2544. IEEE.

[3] J. Balzer and S. Werling. Principles of shape from specular reflection. *Measurement*, 43(10):1305–1317, 2010.

[4] E. A. Bier and K. R. Sloan. Two-part texture mappings. *IEEE Comput. Graph. and Appl.*, 6(9):40–53, 1986.

[5] P. Bovet and S. Benhamou. Spatial analysis of animals' movements using a correlated random walk model. *J. Theor. Biol.*, 131(4):419–433, 1988.

[6] S. Chandrasekhar. Stochastic problems in physics and astronomy. *Rev. Mod. Phys.*, 15:1–89, 1943.

[7] B. Denkena, H. Ahlers, F. Berg, T. Wolf, and H. K. Tönshoff. Fast inspection of larger sized curved surfaces by stripe projection. *CIRP Ann. Manuf. Technol.*, 51(1):499–502, 2002.

[8] U. M. Erdem and S. Sclaroff. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Comput. Vis. Image Und.*, 103(3):156–169, 2006.

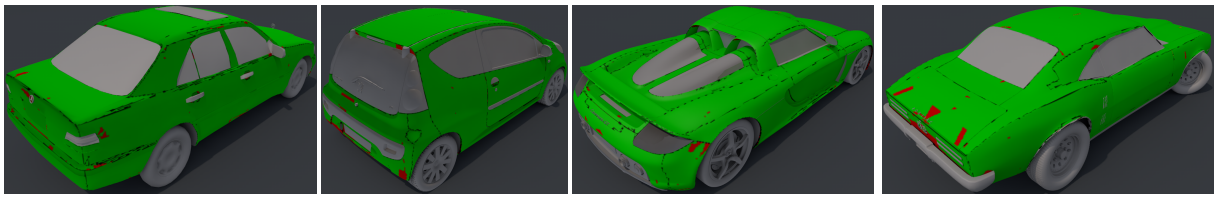


Figure 8: Visualization of the coverage for three of the models used during the optimization. Green is covered, red is not covered, black is not externally visible. The rightmost image shows the coverage of a car that was not part of this set. Nevertheless, we still achieve a good coverage.

[9] O. Faugeras and R. Keriven. Complete dense stereovision using level set methods. In *Computer Vision*, ECCV '98, pages 379–393, 1998.

[10] A. B. Finnila, M. A. Gomez, C. Sebenik, C. Stenson, and J. D. Doll. Quantum annealing: A new method for minimizing multidimensional functions. *Chem. Phys. Lett.*, 219(5–6):343–348, 1994.

[11] S. Fleishman, D. Cohen-Or, and D. Lischinski. Automatic camera placement for image-based modelling. *Comput. Graph. Forum*, 5:231–239, 2000.

[12] L. Grady. Random walks for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(11):1768–1783, 2006.

[13] N. Halper and P. Olivier. Camplan: A camera planning agent. In *Smart Graphics 2000 AAAI Spring Symposium*, pages 92–100, 2000.

[14] D. Hong, H. Park, and H. Cho. Design of a multi-screen deflectometer for shape measurement of solder joints on a PCB. In *Proc. ISIE '09*, pages 127–132. IEEE, 2009.

[15] I. Ihrke, K. N. Kutulakos, H. P. A. Lensch, M. Magnor, and W. Heidrich. Transparent and specular object reconstruction. *Comput. Graph. Forum*, 29(8):2400–2426, 2010.

[16] S. Kammel and F. P. León. Deflectometric measurement of specular surfaces. *IEEE Trans. Instrum. Meas.*, 57(4):763–769, 2008.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[18] M. C. Knauer, J. Kaminski, and G. Häusler. Phase measuring deflectometry: A new approach to specular free-form surfaces. In *P. Soc. Photo-Opt. Ins.*, 2004.

[19] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *Proc. SIGGRAPH '00*, pages 131–144. ACM, 2000.

[20] H. C. Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133–135, 1981.

[21] R. Martoňák, G. E. Santoro, and E. Tosatti. Quantum annealing of the traveling-salesman problem. *Phys. Rev. E*, 70:057701, 2004.

[22] M. Meila and J. Shi. Learning segmentation with random walk. In *Proc. NIPS '01*, 2001.

[23] A. T. Murray and K. Kim. Coverage optimization to support security monitoring. *Comput. Environ. Urban.*, 31:133–147, 2007.

[24] G. Olague and R. Mohr. Optimal camera placement for accurate reconstruction. *Pattern Recognit.*, 35(4):927–944, 2002.

[25] M. Piccardi. Background subtraction techniques: a review. In *Proc. Sys. Man. Cybern.*, pages 3099–3104. IEEE, 2004.

[26] N. Qian. Binocular disparity review and the perception of depth. *Neuron*, 18(3):359–368, 1997.

[27] A. Sanderson, L. Weiss, and S. Nayar. Structured highlight inspection of specular surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(1):44–55, 1988.

[28] H. Schultz. Retrieving shape information from multiple images of a specular surface. *IEEE T. Pattern. Anal.*, 16(2):195–201, Feb 1994.

[29] F. Spitzer. *Principles of random walk*. Springer-Verlag, 2001.

[30] J. Zhao, S. Cheung, and T. Nguyen. *Optimal visual sensor network configuration*. AP, 2009.

[31] J. Y. Zheng and A. Murata. Acquiring a complete 3D model from specular motion under the illumination of circular-shaped light sources. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):913–920, 2000.

A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds

Christian Günther[1] guechr
Thomas Kanzok[1] tkan
Lars Linsen[2] l.linsen
Paul Rosenthal[1] ropau

[1] Chemnitz University of Technology
Department of Computer Science
Visual Computing Laboratory
Straße der Nationen 62
09111 Chemnitz, Germany
[acronym]@hrz.tu-chemnitz.de

[2] Jacobs University
School of Engineering & Science
Visualization and Computer Graphics Laboratory
Campus Ring 1
28759 Bremen, Germany
[acronym]@jacobs-university.de

ABSTRACT

Direct rendering of large point clouds has become common practice in architecture and archaeology in recent years. Due to the high point density no mesh is reconstructed from the scanned data, but the points can be rendered directly as primitives of a graphics API like OpenGL. However, these APIs and the hardware, which they are based on, have been optimized to process triangle meshes. Although current API versions provide lots of control over the hardware, e.g. by using shaders, some hardware components concerned with rasterization of primitives are still hidden from the programmer. In this paper we show that it might be beneficial for point primitives to abandon the standard graphics APIs and directly switch to a GPGPU API like OpenCL.

Keywords

OpenCL, GPGPU, OpenGL, Point Cloud Rendering

1 INTRODUCTION

In architecture as well as archaeology laser scanning has become a valuable tool to capture spacious environments for processing and inspection. The common use cases include airborne Lidar-scanning [RD10] or terrestrial laser scanning systems for urban reconstruction [NSZ+10], virtual inspection of caves and catacombs [SZW09], as well as documentation of excavation sites [LNCV10]. Regardless of the particular application, the scanning process usually produces a dense point cloud, often consisting of several hundred million samples. Those are comprised of a geometric locus and are sometimes enhanced with color or normal information. In order to work with the data in an interactive manner it has to be visualized efficiently, which often involves preprocessing the point clouds to cope with the data size [Lev99, SMK07].

While some approaches aim at the reconstruction of a mesh surface from the data [PV09], several others content with the direct visualization of the point data, either as simple point primitives [WS06] or by using splatting [WBB+08]. Although the latter usually produces renderings of higher quality, there are several preprocessing steps involved, which limit the applicability for direct on-site previews of the captured data.

Raw point primitives on the other hand suffer from the fact that closed surfaces can only be achieved if the sampling density, projected to the screen, exceeds the viewport resolution. However, also in the presence of slightly undersampled data satisfactory depictions can be produced by filtering and post-processing the generated rendering in screen space [KLR12, RL08].

For the actual rendering of its preferred representation virtually every approach uses one of the standard graphics APIs, like DirectX or - in the scientific community predominantly - OpenGL. These APIs are designed to make use of the massive parallel processing power of todays GPUs by following a quite strict pipeline, which all graphics primitives have to traverse before they are displayed to the screen. Although the APIs provide different primitive types to render, what they are optimized for is the primitive type that is commonly used in the consumer market - triangle meshes, like they occur in practically all modern 3D games.

The widespread use of this geometric structure has led to spezialized hardware with dedicated processing units for the different pipeline stages. Although many units are programmable using shaders, some parts of the hard-

ware, in particular the rasterization units, are still not exposed to the programmer. These units, although immensely useful for dealing with large numbers of triangles, do not offer any benefit for direct point rendering, where each point gets projected to exactly one fragment during vertex processing. Unfortunately, they can not be deactivated or circumvented in the current API implementations and therefore form an unnecessary bottleneck for point cloud rendering by restricting the primitive throughput to just a few rasterization processors instead of thousands of shader cores.

Direct access to parallel computing hardware is provided by OpenCL or CUDA, which leads to the question if reimplementing a graphics pipeline specifically tailored towards point primitives could provide significant performance improvements when dealing with this kind of data. In this paper we present such an implementation using OpenCL and show results, which show major speedups compared to the standard OpenGL pipeline in real world datasets.

The rest of the paper is organized as follows: After giving a short overview over the field of point cloud rendering and related work on self-implemented rendering pipelines we describe the challenges and design considerations that went into our implementation. Afterwards we present the implementation of our pipeline and provide a detailed performance analysis on both real and synthetic data.

2 RELATED WORK

Point based rendering is a well studied field in computer graphics and there exists a large amount of literature on that topic. Since this paper's contribution does not lie in any new rendering technique, we give only a short overview of the two main classes of local surface reconstruction methods used in this field and refer the reader to the survey literature [GP07, KB04] for a more in-depth explanation.

A common problem when using point primitives is that, unless the sampling density of the model is really high, there can always be more or less prevalent holes in the rendering. We divide the reconstruction techniques used to produce a hole-free rendering into object space approaches, which require some kind of preprocessing to work, and pure image space approaches, which operate only on the rendering of the raw point cloud.

The most dominant object space approach has arguably become splatting, which was initially introduced for volumetric data by Westover [Wes90]. The approach was later adapted to only use surface samples [PZvBG00, RL00] and it has undergone several extensions and improvements since [GGP04, PSL05, ZPvBG01], even making it applicable for ray tracing [LMR07, SJ00].

However, all these approaches have to compute local surface parameters like splat size, normal direction, curvature etc. in advance. This can take a considerable amount of time when processing really large point clouds as produced by modern laser scanning systems. On the other hand, GPU computing capabilities have increased vastly in recent years, making it possible to compute the splat parameters purely in image space for only those points that are actually visible [PJW12]. When the data is dense enough to only exhibit small holes in the rendering these holes can be filled using interpolation [GD98, PGA11] or special morphological filters [DRL10, RL08].

With triangle rasterization being implemented in graphics cards there has not been much practical need to reimplement the process in software. There are some cases that benefit from a custom rasterizer, although they are mostly limited to applications on gaming consoles [Val11] or to the usage of non-triangular parametric surface patches [Eis09].

Actual GPGPU implementations of triangle rasterizers using CUDA have mainly been developed out of academic interest and as a benchmark application for the current state of GPGPU computing [LK11, LHLW10]. These primarily have to overcome the problem of assigning the triangles to different threads in a way that maximizes the amount of parallel coverage computations for them [MCEF08].

However, their results suffer heavily from the necessary sorting and therefore still do not reach the performance of the hard-wired hardware implementation. Since point primitives only cover one fragment, these prior results are not applicable to our problem. In the following chapters we show that GPGPU-based software rasterization of point clouds is not only not slower than the hardware pipeline, but can yield a significant performance increase with datasets common in architecture and archaeology.

3 DESIGN CONSIDERATIONS

Our goal in this work is to develop a point cloud rendering pipeline that produces results true to the ones obtained when using OpenGL rendering. For best performance in OpenGL, we setup a minimalistic OpenGL 4 pipeline using the early depth test, which saves some fragment shader instantiations under certain conditions. In order to achieve the same results we have to rebuild the basic OpenGL rendering pipeline in GPGPU software. Since we are dealing with zero-dimensional primitives we can omit a tessellation or geometry shader stage (although the latter could be added on demand – provided its output is point primitives again). Also we did not yet include normal information for shading and restricted our first prototype to only use geometry and color information. However, these additions can be easily implemented when needed without major changes in relative render times. The pipeline implemented in our software is shown in Figure 1. We decided to use

OpenCL in favor of CUDA to ensure platform independence. Theoretically, our renderer would not even have to run on a graphics card but could also be used on any other multicore processing unit, e.g. CPUs, hybrids of CPU and GPU like AMDs Accelerated Processing Units [Bro10] or Intels recently introduced Xeon Phi coprocessors [Xeo12].

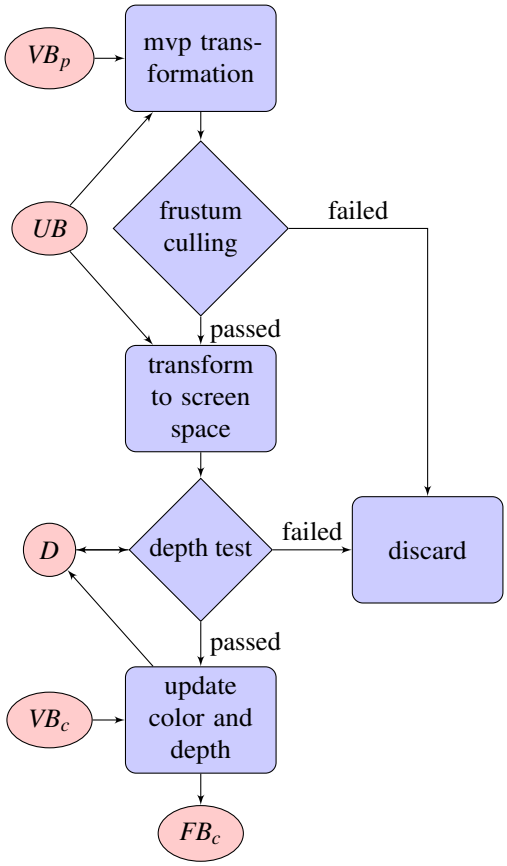


Figure 1: The rendering pipeline implemented by our software. After fetching the vertex coordinates from a vertex buffer object VB_p we project them from object space to clip space via a model-view-projection matrix from a uniform buffer object *UB*. In this space we can already discard many points outside the view frustum. The remaining points are transformed to screen space using the screen resolution from the same uniform buffer object, where their respective positions in the depth buffer *D* is known and can be used to discard occluded points. For the remaining points, we fetch their color information from the vertex buffer object VB_c and write color and depth to the depth buffer and color buffer FB_c, respectively.

To maximize parallel throughput we have to relax the OpenGL paradigm that "Commands are always processed in the order in which they are received. [...] This means, for example, that one primitive must be drawn completely before any subsequent one can affect the frame buffer" [SA12]. One could argue that we consider complete point clouds as one primitive for which the restriction holds. Anyhow, we found that this does not pose a serious limitation since point cloud data from laser scanners does usually not contain transparent points for which the ordering of draw calls would make a difference. What *can* happen is a temporal flickering of points when z-fighting (the term is used here to refer to points that receive the same depth value after projection) occurs under a random draw order. However, this was not noticeable in our experiments.

The central problem of parallel software rendering is to ensure a thread-safe depth test. This test is necessary to guarantee that for each fragment only the point closest to the viewer gets drawn to the frame buffer. To achieve this in a thread safe way, we have to use a global depth buffer that is shared over all compute units of the GPU in connection with atomic operations provided by OpenCL. Unfortunately, as of now it is not possible to use the actual OpenGL depth buffer as shared OpenCL buffer. That is why we allocated our own pure OpenCL buffer for this proof of concept. We could implement a thread-safe depth test on an integer buffer using OpenCL atomics (`atomic_min`). However, this would only work if we wanted to render *only* to the depth buffer. When we also want to write color or other attributes (see Algorithm 1) this approach could lead to a race condition, as depicted in Figure 2.

1: **if** zDepth $<$ atomic_min(depthMap,zDepth) **then**
2: write color information for the current fragment to the color buffer
3: **end if**

Algorithm 1: *Not* thread-safe depth test

To overcome this, we have to expand our critical section (i.e. the section of code, which may not be executed in parallel) to include *all* buffer reads and writes. Such a behavior can be implemented using an atomic flag that indicates whether any thread is currently accessing the section and in this way assures mutual exclusion (mutex). The following section will provide details on the implementation of this solution.

4 IMPLEMENTATION

4.1 Basic Approach

OpenCL offers the possibility to share buffer objects with OpenGL (except depth buffers), which makes it possible to use nearly the same buffer layout for OpenGL and OpenCL rendering. In particular we are using two vertex buffer objects VB_p and VB_c for the point cloud's position (as an array of `float[3]`) and color (as an array of `unsigned char[3]`), respectively. The rendering is done into the color attachment FB_c of a frame buffer object. To transfer the necessary information about the transformation matrices and viewport resolution, we are using a shared uniform buffer object *UB* with the following layout:

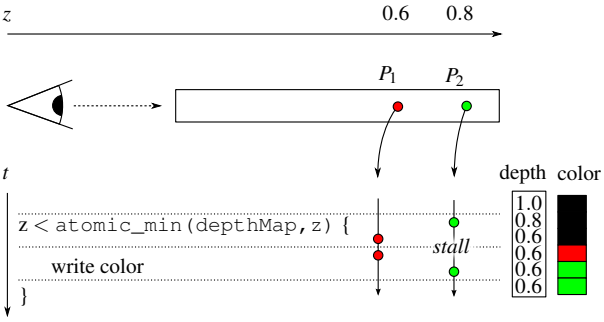


Figure 2: Race condition with a naive depth buffering approach. Assume two points P_1 and P_2 get processed in parallel by the GPU and they are projected to the same fragment. After projection P_1 is closest to the viewer and therefore its color should be written to the color buffer. However, if the first line of Algorithm 1 is processed first for P_2 (letting P_2 pass the depth test for now) and the processing thread is stalled for some reason before line two is reached, then the color of P_1, which passed the depth test in the meantime and was written to the color buffer, would be overwritten without re-checking the depth buffer.

- UB_{MVP} the pre-multiplied model-view-projection matrix
- UB_r the viewport resolution

Additionally, we allocate our own depth buffer D that contains the current closest depth value D_z interleaved with a binary lock D_l for each fragment (see also Figure 1 for the usage pattern of these buffers). The points are processed in parallel, so each work item is responsible for exactly one point.

The thread-safe variant of our rasterization kernel is depicted in Algorithm 2. After extending the vertex coordinates to homogeneous coordinates, applying the standard transformations, and discarding all points that are not in the view frustum (lines 1 to 5), we try to lock the fragment to which our point was projected by using `atomic_cmpxchg` on the lock-component of our depth buffer. If we are able to obtain the lock, we can perform the depth test and write new color and depth information if the test was successful. If we can *not* acquire the lock we have to wait for it to be freed again. This "busy waiting" is a waste of processing time, since many points would probably not pass the depth test in the first place.

For that reason we added an early-out mechanism by enclosing the whole critical section into the else branch of an upstream z-test (see Algorithm 3).

The possibility to discard points early that would not influence the final rendering causes major speedups in cases where there are lots of points projected to one fragment.

1: $\mathbf{p} \leftarrow VB_p(i)$ {load one point position $\mathbf{p}$ per thread i, append homogeneous 1}
2: $\mathbf{p} \leftarrow UB_{MVP} \cdot \mathbf{p}$ {transform to clip space}
3: perform frustum culling
4: $\mathbf{p} \leftarrow \mathbf{p} \cdot \frac{1}{p_w}$ {transform to normalized device coordinates}
5: $\mathbf{p} \leftarrow (\mathbf{p} + [1,1,1,1]^T) \cdot 0.5 \cdot [UB_{r.x}, UB_{r.y}, 1, 1]^T$ {transform to screen space}
6: **while p** not processed **do**
7: try to lock $D_l(p_x, p_y)$ with `atomic_cmpxchg`
8: **if** got the lock **then**
9: **if** $p_z < D_z(p_x, p_y)$ **then**
10: $D_z(p_x, p_y) \leftarrow p_z$
11: $FB_c \leftarrow VB_c(i)$
12: **end if**
13: mark as processed
14: free the lock $D_l(p_x, p_y)$
15: **end if**
16: Barrier
17: **end while**

Algorithm 2: The basic point rasterization algorithm with thread-safe depth test.

1: ...
2: **while p** not processed **do**
3: **if** $p_z \geq D_z(p_x, p_y)$ **then**
4: mark as processed
5: **else**
6: [lines 7 to 16 of Algorithm 2]
7: **end if**
8: **end while**

Algorithm 3: The early-out optimization for the thread-safe depth test of Algorithm 2.

4.2 Challenges and Solutions

During implementation we noticed several pitfalls and shortcomings of current drivers and the OpenCL API, which we had to find workarounds for. In this section we will present the problems we found and explain how we were able to solve them.

4.2.1 Implicit Compiler Optimizations

The OpenCL compiler, which is included in the vendors graphics card driver, performs lots of implicit optimizations on the code. It reorders the OpenCL code to achieve the highest possible instruction level parallelism. This is achieved by analyzing the code, especially finding reads and writes to the same memory locations and evaluating dependencies in computations. Unfortunately for a mutex structure, the actual critical section works on completely different memory than the lock, which can be missed by the compiler. To ensure correct locking behavior, we have to insert a memory barrier in our code. This barrier ensures that the reordering does not exceed this point during compilation *and* run time.

4.2.2 Accuracy

Strict frustum culling is essential for the software rendering. One has to make sure to only read or write in the valid ranges of the depth and color buffers (otherwise the graphics driver may freeze). Unfortunately, it seems that unsafe internal math optimizations can lead to out-of-bound buffer accesses when only doing the culling in clip space. Therefore we have to add an additional range check after transforming the coordinates to the viewport.

In addition, the performance of our approach can receive a huge performance boost when using the full floating point range for depth testing. Normally, OpenGL takes normalized real depth values in the range of $[-1,1]$ from the normalized device coordinates (after the division by w) of a point, maps them to $[0,1]$, and stretches this interval to a 24 bit integer which is used for the depth test. Our implementation uses a floating point buffer with 32 bit precision. Mapping these depth values to the $[0,1]$ interval may introduce discretization errors in the binary representation that can be alleviated when omitting the division by w for the depth value.

This does not lead to inconsistent depth values as it would when using triangles, since we do not have to interpolate this value in image space.

The described procedure does not only lead to a significant boost in performance, due to the more efficient early-out mechanism (see Table 1), but also eliminates the occasional depth flickering which was noticeable with the previous approach.

4.2.3 Depth Sharing

Sharing an OpenGL depth buffer with OpenCL is still not supported in current drivers, although there are efforts in this direction via a proposed extension [CLE12]. As of now, one has to render the depth buffer to the OpenGL depth buffer in a consequent pass. Since the extension specification was published in November 2012, we are confident that future drivers will support shared depth buffers, which makes it easier to integrate OpenCL rendering into existing engines.

4.2.4 Caching Effects

Normally the driver would cache read and write operations to buffer objects, which is generally not a problem, since most buffers are either read-only or write-only. For our depth buffer, however, we need reading as well as writing in combination with atomic operations. This pattern can lead to problems when caches of several compute units are involved in the computation. In fact, it can happen that one thread uses a value from its cache while the actual depth value in the buffer has already changed. To overcome this problem, the depth buffer has to be declared `volatile`, causing the driver to broadcast changes to every compute unit that uses the buffer.

5 RESULTS AND DISCUSSION

To evaluate the performance of our approach, we performed benchmarks on a synthetic and a real-world dataset. The synthetic one enables us to evaluate the two determining factors – the number of points in the dataset and the number of z-tests in the depth buffer – in detail. The real-world scan provides a direct comparison between a minimalistic OpenGL 4 pipeline with early depth test and our own OpenCL rendering pipeline.

For our experiments we used a viewport with a size of 1024×1024 pixels and designed the synthetic dataset to align with the viewport pixels when using parallel projection. For now we focused our experiments on AMD hardware, i.e. the Radeon HD 7970 GPU, because it supports all modern OpenCL features and offers the best balance between large memory – useful for experiments with large amounts of data – and an affordable price. Nevertheless we also validated our results with an Nvidia GeForce GTX 680 graphics card.

5.1 Synthetic Data

The synthetic dataset was created as a cuboid of n planes with 1024^2 points each. This facilitates direct control over the number of z-tests during rendering. The number of planes was chosen to be the largest possible such that the entire dataset fitted into the 3GB of GPU memory. To eliminate possible effects caused by the structure of the data, all vertices were shuffled in memory to ensure a random point distribution. Finally, smaller subsets with $k < n$ planes were created to be used for incremental growth of the dataset during testing.

The left part of Figure 3 shows the rendering performance in milliseconds per frame while we were gradually increasing the number of points in the dataset. In each step one plane of 1024^2 points was added to the cuboid, effectively increasing the number of z-tests per fragment by one. The benchmarks indicate that our pipeline is – with a rendering speed of 4.47ms vs. 2.24ms per frame – slightly slower than the OpenGL implementation when there are few z-tests in the image. As the number of z-tests increases the speed of OpenCL rendering compared to OpenGL rendering grows linearly until the OpenCL rendering is as fast as the one with OpenGL at 46 z-tests per fragment.

When the full GPU memory capacity was reached with 170 planes (around 178 million points) the speed compared to OpenGL reached 121%. Afterwards we had to rearrange the dataset to increase the number of z-tests. We decreased the base resolution until the whole dataset lay in one single pixel. The performance results of this step can be seen in the right part of Figure 3. The OpenGL rendering times are still increasing while our pipeline gets even faster with more z-tests, because large amounts of points can be discarded early.

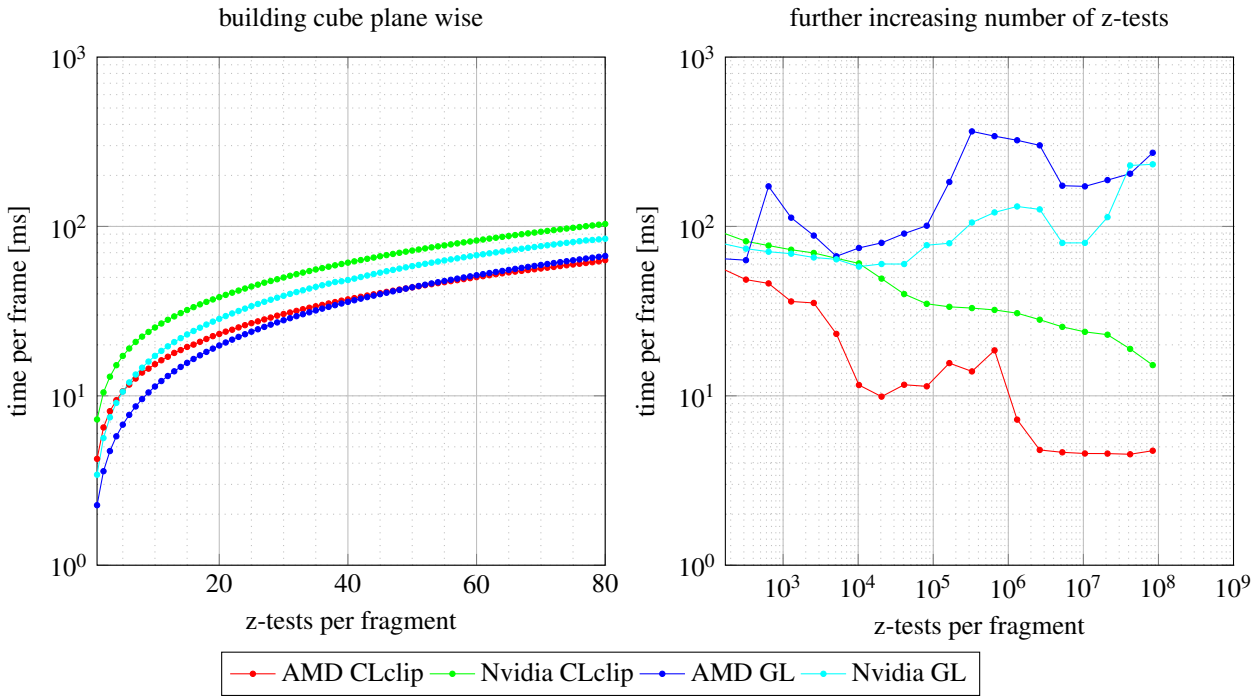


Figure 3: Rendering time in milliseconds (on a logarithmic scale) for different dataset sizes. The left plot shows the results of gradually increasing the number of planes in a 1024^2 cuboid, leading to increased amount of necessary z-tests. In the right plot the overall size of the dataset stays the same, while we gradually rearranged it to cover fewer pixels. In each step the effective resolution was halved in alternating directions, leading to a doubling of the amount of z-tests. In the end the whole dataset is projected to one single pixel. The timings represent the median of 100 distinct measurements. The flattening in the end of the OpenCL series for the Radeon HD 7970 is probably due to efficient caching in local memory (It begins at 32 covered fragments which corresponds to one fragment per each of the 32 compute units).

5.2 Real World Data

To compare the real-world performance of our rendering approach with the fixed-function one, we use a real laser scanning dataset with 138 million points. The data was obtained from a bridge and is composed of five single scans registered into one dataset.

In Figure 4 we show a comparison of the rendering results using OpenGL and OpenCL for the same view of the dataset. There are no noticable differences in rendering quality, but the performance of the OpenCL renderer was more than ten times as high as the OpenGL performance (18ms vs. 129ms or 56fps vs. 5fps, respectively). This can be attributed to the immensely large number of z-tests, especially in the vicinity of the scanners (see Figure 5).

To further analyze these results, we also measure the rendering performance of our approach from a variety of different representative viewpoints in the scene. The results, which can be seen in Table 1, show that the OpenCL renderer outperforms the OpenGL one in all given situations. Although Nvidia's driver support for OpenCL lacks the efficiency of their CUDA one, we also tested the same situations on a GeForce GTX680. Because Nvidia's OpenCL driver does not allow for sharing larger buffer objects, we had to downsample the dataset to half its original size. In this test we were able to achieve better performance in most cases, while the complete overview with plenty of z-tests was faster in OpenGL when using normalized depth values. We assume that our early depth test does not perform as good on Nvidia's hardware as on the one by AMD. However, when doing the depth test in clip space this is accelerated by a factor of 5. In any case the whole pipeline is faster in common use cases.

It can, however, only reach its full potential, when there is a large amount of z-tests. This makes it difficult to make any assumptions as to how OpenCL rendering could influence existing point cloud rendering systems [RD10, WBB+08, WS06], since they all incorporate some level-of-detail mechanism, which inherently minimizes the amount of z-tests. What *could* be done in such a case to further improve performance in our renderer, is to superimpose some space partitioning structure to enable early frustum culling. We left this for future work, since we wanted to investigate the theoretical possibilities of the brute-force method first.

(a) (b)

Figure 4: Comparison of (a) OpenGL rendering and (b) OpenCL rendering. As one can see, rendering results are almost identical. There are only small differences in far away parts of the scene, because we use eight bits more depth buffer precision than OpenGL. While the OpenGL version runs at about 5 frames per second, our implementation reaches 56 fps.

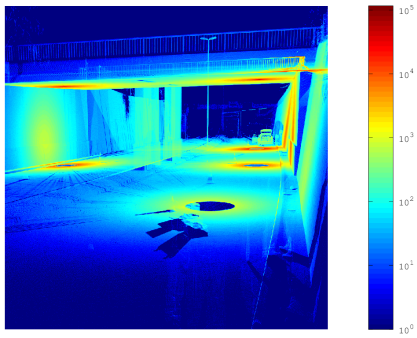


Figure 5: An overview of the amount of z-tests in the view of Figure 4. The scale is logarithmic and ranges from 1 (blue) to 116243 (red) points per fragment.

6 CONCLUSIONS

We have presented an implementation of a point rendering pipeline using OpenCL. Although the approach is certainly not completely optimized yet, point cloud rendering in OpenCL already promises huge gains in rendering time compared to OpenGL. We were able to achieve speedups of one order of magnitude for typical datasets and view parameters. Hierarchical frustum culling and clever reordering strategies on the GPU could push the boundaries even further.

There are several points we had to postpone for now, because the current specifications of OpenCL do not account for the desired behavior. If those possibilities were to be added in the future we would like to include them in our implementation. First of all, using a shared depth buffer will be essential for combined OpenCL-OpenGL rendering. Although the necessary extension has been specified, we have to wait for hardware vendors to implement it in their drivers. Also it would be great (but is not planned at the moment) if a possibility existed to stall waiting threads directly from the OpenCL code. That way waiting threads could be rescheduled for a while, letting the respective compute unit process other points while some fragment is locked. Third, a further investigation of register usage by the OpenCL compiler could provide helpful insights. There seems to be quasi-random utilization of GPU registers by the OpenCL compiler, even when using very strict scopes for variables. This leads to significant performance differences when compiling nearly identical code on the same or on different platforms. Overcoming this problem seems to be impossible with the current drivers

7 ACKNOWLEDGMENTS

The authors would like to thank the enertec engineering AG (Winterthur, Switzerland) for providing us with the data and for their close collaboration. This work was partially funded by EUREKA Eurostars (Project E!7001 "enercloud - Instantaneous Visual Inspection of High-resolution Engineering Construction Scans").

8 REFERENCES

[Bro10] Nathan Brookwood. Amd Fusion^{TM} Family of Apus: Enabling a Superior, Immersive PC Experience. Technical report, Advanced Micro Devices, Inc., March 2010.

(a) View 2 (b) View 3 (c) View 4

	Point distribution CL_c			Radeon HD 7970					GeForce GTX 680				
View	Culled	Early Out	Z-Test	GL	CL_n	GL/CL_n	CL_c	GL/CL_c	GL	CL_n	GL/CL_n	CL_c	GL/CL_c
1	6.50	128.78	3.26	5.21	58.68	11.27	61.46	11.80	6.98	19.24	2.76	23.26	3.33
2	8.70	126.24	3.62	5.73	55.04	9.61	59.91	10.45	6.97	19.73	2.83	23.96	3.43
3	130.57	5.31	2.67	13.14	97.56	7.43	102.50	7.80	7.04	77.29	10.97	78.95	11.20
4	0	138.07	0.48	2.91	15.36	5.28	90.66	31.14	6.22	2.13	0.34	36.38	5.84

Table 1: Rendering performance for some representative points of view in frames per second. Compared is the rendering performance of OpenGL (GL) and OpenCL with depth test in clip space (CL_c) or in the space of normalized device coordinates (CL_n). The views include large amounts of z-tests in combination with some frustum culling (View 1, see Figure 4, and View 2), intense frustum culling with few z-tests (View 3) and an overview of the complete dataset with an enormous amount of z-tests and no frustum culling (View 4). The same experiments were carried out on an Nvidia GeForce GTX680 (right). Unfortunately, we had to reduce the amount of points to 50% here, because the driver did not allow the sharing of larger buffer objects. Additionally we measured the amount of points that were processed in the different stages of our pipeline (all values are given in millions). The exact relation between early out and z-test depends on scheduling and latencies, which are slightly influenced by the measurement itself. Summing up the points that were culled, rejected by the early-out mechanism and finally processed by the depth test gives the dataset size of 138 million Points.

[CLE12] The OpenCL Extension Specification Version 1.2. Technical report, Khronos OpenCL Working Group, November 2012.

[DRL10] Petar Dobrev, Paul Rosenthal, and Lars Linsen. Interactive Image-space Point Cloud Rendering with Transparency and Shadows. In Vaclav Skala editor, *Communication Papers Proceedings of WSCG, The 18th International Conference on Computer Graphics, Visualization and Computer Vision*, pages 101–108, Plzen, Czech Republic, 2 2010. UNION Agency–Science Press.

[Eis09] Charles Loop Christian Eisenacher. Real-time patch-based sort-middle rendering on massively parallel hardware. Technical report, Microsoft Research, May 2009.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques 98*, pages 181–192. Springer, 1998.

[GGP04] Loïc Barthe Gaël Guennebaud and Mathias Paulin. Deferred splatting. *Computer Graphics Forum*, 23(3):653–660, 2004.

[GP07] Markus Gross and Hanspeter Pfister, editors. *Point-Based Graphics*. Morgan Kaufmann, 2007.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers &; Graphics*, 28(6):801 – 814, 2004.

[KLR12] Thomas Kanzok, Lars Linsen, and Paul Rosenthal. On-the-fly luminance correction for rendering of inconsistently lit point clouds. *Journal of WSCG*, 20(2):161 – 169, 2012.

[Lev99] M. Levoy. The digital michelangelo project. In *3-D Digital Imaging and Modeling, 1999. Proceedings. Second International Conference on*, page 2–11, 1999.

[LHLW10] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM.

[LK11] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM.

[LMR07] Lars Linsen, Karsten Müller, and Paul Rosenthal. Splat-based ray tracing of point clouds. *Journal of WSCG*, 15(1-3):51–58, 2007.

[LNCV10] José Luis Lerma, Santiago Navarro, Miriam Cabrelles, and Valentín Villaverde. Terrestrial laser scanning and close range photogrammetry for 3D archaeological documentation: the upper palaeolithic cave of parpalló as a case study. *Journal of Archaeological Science*, 37(3):499–507, March 2010.

[MCEF08] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 35:1–35:11, New York, NY, USA, 2008. ACM.

[NSZ+10] Liangliang Nan, Andrei Sharf, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. SmartBoxes for interactive urban reconstruction. In *ACM SIGGRAPH 2010 papers*, page 93:1–93:10, New York, NY, USA, 2010. ACM.

[PGA11] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. Real-time rendering of massive unstructured raw point clouds using screen-space operators. In Franco Niccolucci, Matteo Dellepiane, Sebastián Peña Serna, Holly E. Rushmeier, and Luc J. Van Gool, editors, *VAST*, pages 105–112. Eurographics Association, 2011.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[PSL05] Renato Pajarola, Miguel Sainz, and Roberto Lario. Xsplat: External memory multiresolution point visualization. In *IASTED International Conference on Visualization, Imaging and Image Processing*, JUL 2005.

[PV09] Shi Pu and George Vosselman. Knowledge based reconstruction of building models from terrestrial laser scanning data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 64(6):575–584, November 2009.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RD10] Rico Richter and Jürgen Döllner. Out-of-core real-time visualization of massive 3D point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, page 121–128, 2010.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[RL08] Paul Rosenthal and Lars Linsen. Image-space point cloud rendering. In *Proceedings of Computer Graphics International*, pages 136–143, 2008.

[SA12] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 4.3 (Core Profile)). Technical report, The Khronos Group Inc., August 2012.

[SJ00] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 319–328, 2000.

[SMK07] Ruwen Schnabel, Sebastian Möser, and Reinhard Klein. A parallelly decodeable compression scheme for efficient point-cloud rendering. In *Proceedings of Symposium on Point-Based Graphics*, page 214–226, 2007.

[SZW09] Claus Scheiblauer, N. Zimmermann, and Michael Wimmer. Interactive domitilla catacomb exploration. In Kurt Debattista, Cinzia Perlingieri, Denis Pitzalis, and Sandro Spina, editors, *VAST*, pages 65–72. Eurographics Association, 2009.

[Val11] Michal Valient. Practical occlusion culling in Killzone 3. Siggraph2011, 2011.

[WBB+08] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Aarno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Processing and interactive editing of huge point clouds from 3D scanners. *Computers & Graphics*, 32(2):204–220, 2008.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 367–376, New York, NY, USA, 1990. ACM.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *Proceedings Symposium on Point-Based Graphics 2006*, page 129–136, 2006.

[Xeo12] Intel® Xeon Phi™ Coprocessor. Technical Report 328209-001EN, Intel Corporation, November 2012.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 371–378, New York, NY, USA, 2001. ACM.