MM Science Journal | www.mmscience.eu
ISSN 1803-1269 (Print) | ISSN 1805-0476 (Online)
Special Issue | HSM 2023
17th International Conference on High Speed Machining
October 25-28, 2023, Nanjing, China
DOI: 10.17973/MMSJ.2023_11_2023112

HSM2023-00049

IMPLEMENTING OF PHOTOREALISTIC SURFACE RENDERING IN CAM SIMULATION

T. Beer[1], M. Stautner[1,2], X. Beudaert[3], M. Gil[3], Z. Dombovari[4], D. Plakhotnik[1,5*]
[1]ModuleWorks GmbH, Aachen, Germany
[2]Hochschule Ruhr West, Muehlheim, Germany
[3]IDEKO S Coop: Elgoibar, Spain
[4]Budapest University of Technology and Economics, Budapest, Hungary
[5]University of Twente, Enschede, the Netherlands
*Corresponding author; e-mail: denys@moduleworks.com

Abstract
In this paper, different approaches to visualize CAM simulation results are presented. Simulated machined surfaces can be rendered in a photorealistic manner taking into account real tool motions and vibrations measured during the machining process. The actual tool trajectories are used to derive the microrelief of machined surface that lead to different reflective properties to be used in the CAM surface rendering pipeline. Broaching and grinding were two machining processes to adopt this photorealistic visualization. First, the generic CAM simulation computed the triangulated representation of the in-process workpiece. Second, the triangles were enriched with the computed micro topography information that is used by the developed shader implementation during the visualization process.

Keywords:
Computer Aided Manufacturing, CAM, Simulation, Grinding, Broaching, Visualization, Rendering, Vibration, Digital Twin

1 INTRODUCTION

Nowadays, the manufacturing industry relies on the use of CAM simulation and NC verification before actual production. CAM simulation has proven to foresee and prevent substantial failures during machining, such as collisions and gouges between the tool and workpiece. However, CAM simulation software cannot grasp on the issues arising during the machining process. For instance, chatter issues may result in severe deterioration of the machined surface quality. Some machining processes, as ball-end milling, generate rough surfaces even without any chatter. Eventually, CAM users do request to include the surface roughness simulation into CAM software, but such a development is not trivial.

[Liu 2005] addressed the visual surface appearance of parts machined by a rotating, multi-flute, ball nose milling cutter. It was an attempt to consider the micro pattern pertaining to the cutting action of individual cutter flutes.

[Bilalis 2009] also, like [Liu 2005] determined the machined surface topography as a cloud of points retrieved from the visualization system Z buffer to calculate surface roughness. Different roughness metrics were calculated from the simulated surface, and isolines of the surface properties were depicted with different colors on the surfaces.

[Wang 2016] simulated surface scallop topology after five-axis milling. Simulation was performed microscale to identify roughness properties. Then, the set of these properties was assigned as a color code to the entire surface machined during the operation.

[Klimant 2014] described a simulation method for NC programs. They used real axis values from a real CNC to simulate the milling process by means of a CAD kernel. The resulted CAD geometry was used for virtual metrology. Part visualization was the conventional CAD visualization with the uniform color.

[Brecher 2017] presented an advanced virtual environment to predict part shape distortions and combine them with the surface quality measurement to show with different color codes.

The objective of this paper is to explore several approaches to define advanced visualization techniques suitable for digital twins aiming to predict workpiece quality. At present, most of the quality controls are performed after the part has been manufactured, which can lead to rework operations, thus increasing the cost of time and machines, or even more importantly, a total refusal of the part if the defects are irrecoverable. With the quality oriented digital twins, the problem of quality control must be addressed before and during the manufacturing process. Existing metrics to assess the surface quality are not enough to comprehensively qualify the part, and in some cases the part must be visually inspected by a naked eye to evaluate the reflective properties of machined parts.

Multiple machining processes – broaching, cylindrical grinding and, to some extend, turning – were considered for implementing real time visualization of the metallic reflectivity and the surface visual artifacts (grooves, etc.).

2 WORKPIECE DATA

The rendering approach we are going to present uses a discrete, triangulated workpiece model as input. We create it with a 3D removal simulation kernel that has the capability to attach additional information to the workpiece surface mesh. With this, we are able to distinguish surface areas that have been machined with different tools from each other. We will see later how this is utilized for the some of the workpiece visualization techniques.

Most of the commercial CAM software use tri-dexel data model [Benouamer 1997] that defines volumes in a discrete manner. As shown in Fig. 1, a solid model can be represented by several linear segments that are aligned to X, Y, and Z axes. The tri-dexel model consists only the linear segments, basically the end points of each segment. Information at every point is enriched with additional properties, like the surface normal and tool move number. The surface of the solid model is not stored in the tri-dexel grid (field). In order to be able to visualize the surface of the part, an additional algorithm must reconstruct the local surface patches between end points of dexels considering their spatial coordinates and surface normal vectors at these points.

The tri-dexel model is proven to curb memory consumption in contrast to triangulated surface. During milling simulation, the storage space is expected to grow relatively moderately regardless the complexity of the simulated part. However, this advantage is due to a trade-off, that some geometric features, which are smaller that the distance between neighbor dexels, may not be spotted by the simulation.

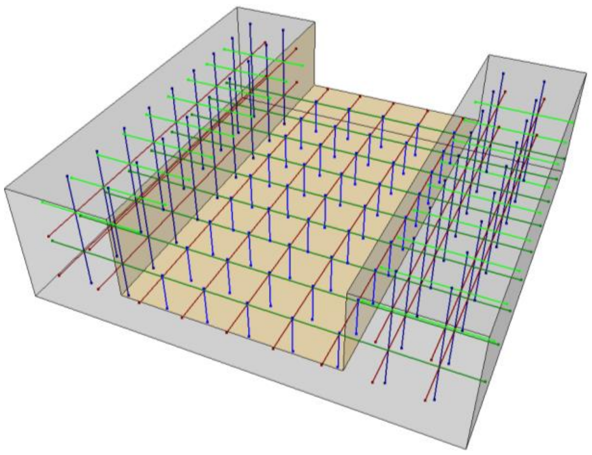


Fig. 1: Solid (semi-transparent) and tri-dexel models (red-green-blue line segments)

3 REAL TIME RENDERING

The visualization techniques presented in this article are all implemented in a real time rendering environment utilizing C++ and OpenGL. In OpenGL version 4, the shader pipeline looks like Fig. 2. Only the Vertex and Fragment Stages must be programmed always. The other shaders (round edged items in Fig. 2) are optional and the remaining stages are only configurable, but not fully programmable.

A shader pipeline is very flexible and allows for a lot of creativity and customizability. One caveat though is, that due to this customization of shader code to a specific application's needs, it is usually not possible to reuse shader code from a different application or from learning resources directly. This is one reason why we do not use third-party visualization or rendering toolkits. That way we are more flexible and in full control of the pipeline and can especially decide whether it makes more sense to modify the input data structures (application side) or the shader programs to realize new ideas.

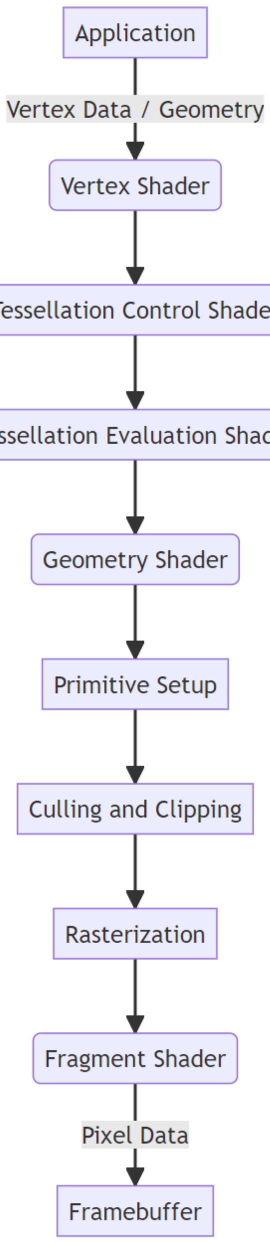


Fig. 2: The OpenGL Rendering Pipeline.

Another downside of a programmable rendering pipeline is that one must take care of literally everything. Not even transformation of the input geometry or basic lighting calculations are provided. It boils down to one basic question that the rasterizer will pass on to the Fragment

Shader in the last rendering stage: 'should this fragment be skipped or does it have a color, and if so, which'. From there one needs to develop through the shader stages, in order to gather and forward all the information needed at that last pipeline stage, to answer that question. Performance-wise it is best to calculate stuff as early in the pipeline as possible, as every successive stage will potentially fan out multiple operations for each single input value.

For the ease of understanding, we named the output data in Fig. 2 'Pixel Data'. Though this is technically not exact, the difference between a fragment and a pixel does not matter for the contents of this article. A 'fragment' can be thought of as a single pixel of the final image on the screen, for now.

So, a lot of the basic steps, e.g., transformation and perspective projection of the incoming workpiece geometry according to the virtual camera position, and especially lighting and shading, must be programmed explicitly. This renders all the literature from the early ages of computer graphics relevant again, because those things had been hidden under the hood of graphics programming APIs and GPU hardware for decades. Since the early ages of computer graphics in the 1970s, technology has changed a lot, but the math behind it has obviously not. Now those resources are still very valid and surprisingly sufficient as a starting point for modern shader programming.

We use the standard Phong reflection model [Phong 1975] to calculate colors for the effect of light sources in the scene (to be correct here, the simplification to directional light sources we are using is also known as the Blinn-Phong reflection model [Blinn 1977]). Basic lighting and material appearance calculations are performed with this model. This gives the 'standard' 3D real time rendering look as a base for the following 'add-on' visual effects.

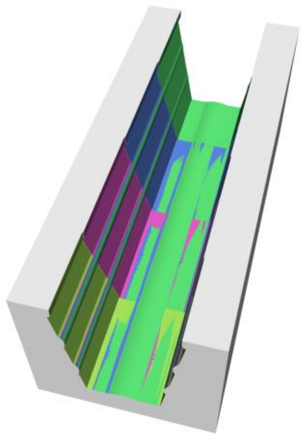
4 BROACHING

For the simulation of a broaching process, we have split the broach tool into single teeth (Fig. 4). The cutting simulation kernel then is driven with each tooth modeled as a different tool. This gives us the opportunity to mock micro vibrations by slightly modifying the tool path of each individual tool/tooth. Although the resolution of the mesoscopic simulation model is not high enough to provide accurate microscopic surface properties from that, it does provide us with the information about which tooth of the broaching tool did remove material at which surface positions. From that we can apply different visual properties to those areas.

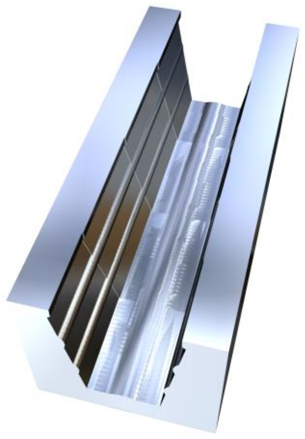
The first implementation provided classical means to assign material properties for intact surfaces along with customized properties for surfaces created by each individual tooth, as shown in Fig. 3. Here, the workpiece surface basically appears colored by tooth, along with standard lighting.

A next version of the visualization involves environmental texture mapping to create a metallic reflective look (Fig. 3 (2) and (4)). The implementation uses the traditional method as described in [Blinn 1976]. It is the same concept that was used in the OpenGL fixed function pipeline. Now that we have resembled this as GLSL shader code, we can very easily add some modifications for more realism.

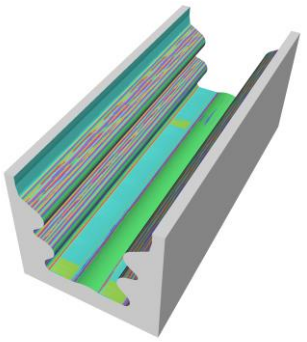
For recreation of the visual effect of different degrees of surface roughness, we perturbate the workpiece surface normal vectors (per fragment) which are used in that reflection calculation. For that we use a concept that is known as 'Perlin noise' [Perlin 1985], see Fig. 5. This type of noise has properties that are substantial for that use



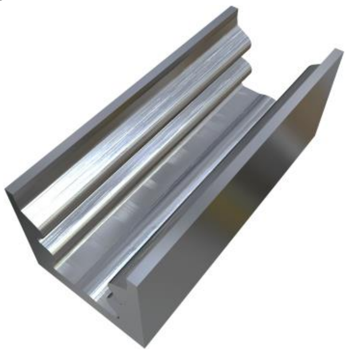
a) After first four teeth engaged, colored by the tooth number



b) After four teeth engaged, with reflections



c) At the end of broaching, colored by the tooth number



d) At the end of broaching, with reflections

Fig. 3: Visualization of the broaching simulation with vibrations.

case: ‘It provides a space filling signal that has an impression of randomness while being controllable, with no high or low spatial frequencies.’ This results in a smooth a nd natural appearance. In fact, normal perturbation was one of the original use cases for which Ken Perlin developed that approach for application in the movie ‘Tron’ in 1982. The idea of normal perturbation per fragment itself, as a means to increase visual detail beyond the geometric data, is even older and traces back to James Blinn in 1978 [Blinn 1978]. Of course, though the math behind the ideas have not changed, the way those are actually implemented has

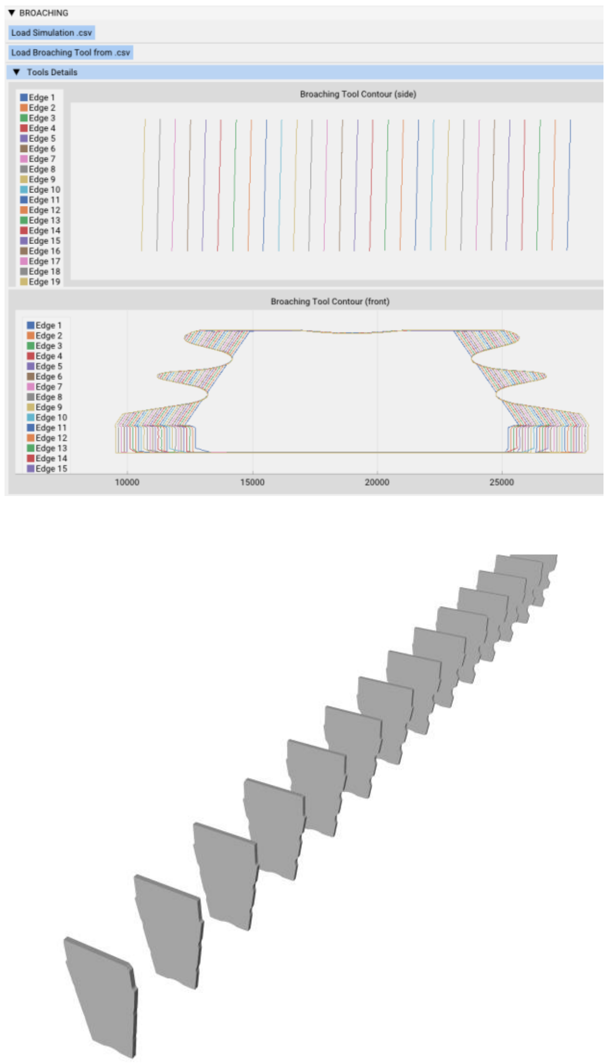


Fig. 4: Tree-shaped broaching tool.
Top: Edge profiles in interactive 2D plot.
Bottom: Generated 3D tooth shapes.

drastically changed.

Our Perlin noise signal is procedurally generated and

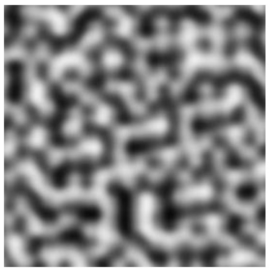


Fig. 5: Typical Perlin Noise.

evaluated on the GPU on the fly, just for the required surface points. We adopted the basic methods presented in [Gustavson 2022] for our noise functions.

We want to create different appearances for smooth or diffuse reflecting surface areas. This is achieved by modulating the base frequency and amplitude of the noise signal, depending on the material properties that are assigned to the specific broaching tool tooth that has touched that surface as last. For example, a perfectly smooth surface will have perfect, crisp reflection properties. This can be achieved by not applying any normal perturbation but straightly reflecting in the direction of the workpiece surface. For a smooth but slightly diffuse appearance we increase the amplitude of the noise. For rough appearance we increase amplitude and frequency.

Technically, we have implemented this by grouping the workpiece triangles by tooth as we receive them from the 3D cutting simulation. Each group is assigned a separate material which contains a base color along with other parameters for the lighting calculations and for shaping the noise signal. There is also a blending factor to control how much of the lighting calculation results and how much of the reflection calculation results contribute to the final fragment color.

In the rendering loop we then process those groups separately: The material parameters are transferred into appropriate shader program variables, then the group of triangles is rendered.

The resulting visualization will then appear with different color/reflection properties depending on the broaching tool tooth that was engaged on that surface point (Fig. 6).

Different surface colors correspond to different teeth that cut the surfaces.

5 CYLINDRICAL GRINDING AND TURNING

Grinding or turning of rotation symmetrical parts has basically a toolpath consisting of helical polylines. The

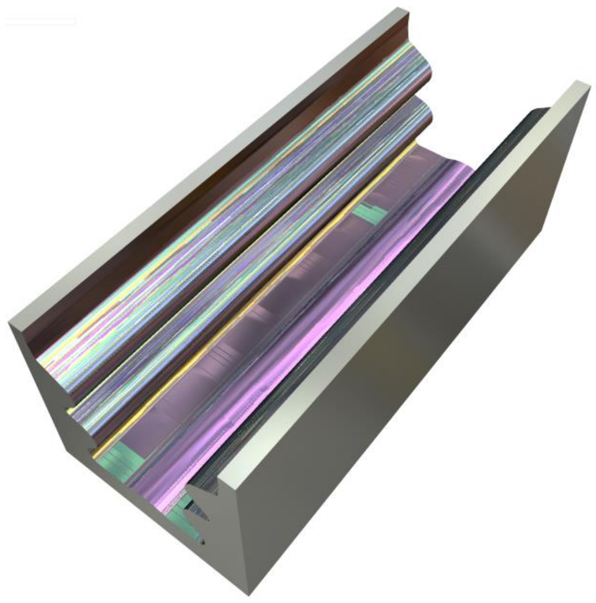
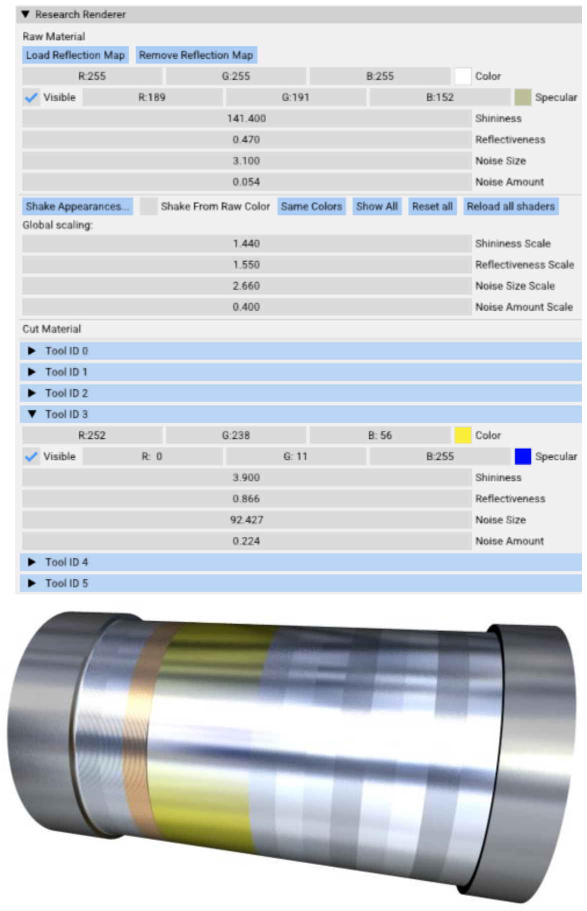


Fig. 6: Workpiece after broaching, with reflection effects and colored by engaged tooth.

points of this polyline may not be exactly on that helix because of the tool and workpiece vibrations. These vibrations cause marks on the surface. Even though the 3D simulation is not appropriate for visualization on the microscopic level, the 3D simulation can still be used with mesoscopic resolution, in order to generate plausible visualization for quick evaluation of foreseen results, as shown in Fig. 8 depicting how visually appealing rendering can be achieved via applying different visual material properties. In Fig. 8 and Fig. 8 the Perlin noise based normal perturbation approach, as described in the previous Broaching section of this article, has been applied. The stronger grooves on the left side are simulated helix cuts and are part of the mesoscopic material removal simulation results. Different parts of the workpiece have been simulated with different tools so that the workpiece model contains different visual materials which can be modified interactively. This approach delivered some appealing 'eye candy' and plausible pictures for the results of a grinding process. But for gaining insight into actual results of simulated machining processes we now will involve more specific information, in order to relate the visual effects to the actual vibration phenomena we are simulating, instead of applying only some controlled noise.

For the visualization of turning process results, we use micro profiles which contain the surface topography errors as axial profiles along discrete angles of the workpiece. The application prototype allows to plot them as 2D graphs for interactive inspection (Fig. 9). They are given as lists of depth offsets perpendicular to the workpiece surface, along the part, at different angular positions.



▼ Research Renderer

Raw Material

Load Reflection Map | Remove Reflection Map

	R:255	G:255	B:255	Color
✓ Visible	R:189	G:191	B:152	Specular
141.400				Shininess
0.470				Reflectiveness
3.100				Noise Size
0.054				Noise Amount

Shake Appearances... | Shake From Raw Color | Same Colors | Show All | Reset all | Reload all shaders

Global scaling:

1.440	Shininess Scale
1.550	Reflectiveness Scale
2.660	Noise Size Scale
0.400	Noise Amount Scale

Cut Material

▶ Tool ID 0

▶ Tool ID 1

▶ Tool ID 2

▼ Tool ID 3

	R:252	G:238	B: 56	Color
✓ Visible	R: 0	G: 11	B:255	Specular
3.900				Shininess
0.866				Reflectiveness
92.427				Noise Size
0.224				Noise Amount

▶ Tool ID 4

▶ Tool ID 5

Fig. 7: Interactive material editor with parameters for each simulated tool.

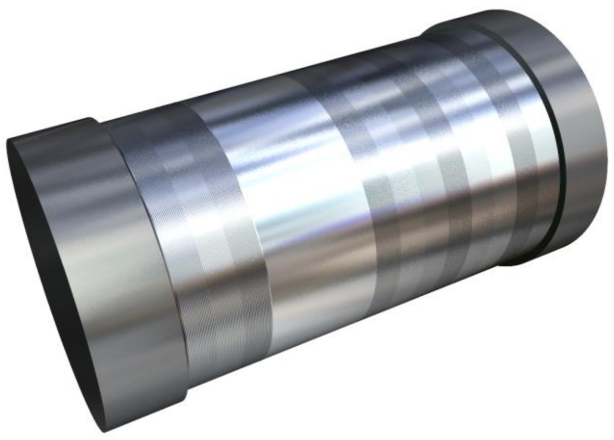


Fig. 8: Cylindric workpiece with surface visualization trying to replicate effects of probing different grinding tools.

The plot shows unitless error magnitude (Y axis), which can be scaled respectively to calibration measurements or visualization purposes, along the sampling points laying on part’s rotary axis (X axis). In this example, there were 359 profiles (angular step 1.002786 degrees) with 1000 data points each, covering a workpiece of 100mm. It depends on the concrete profiles, how many are needed to have enough data so that a special phenomenon becomes visible. E.g. Nyquist-Shannon theorem gives an idea about what order of magnitude of data points is needed to represent certain features in general. Technically, any number of data points and interpolated intermediate points can be taken, as long as there are enough computational

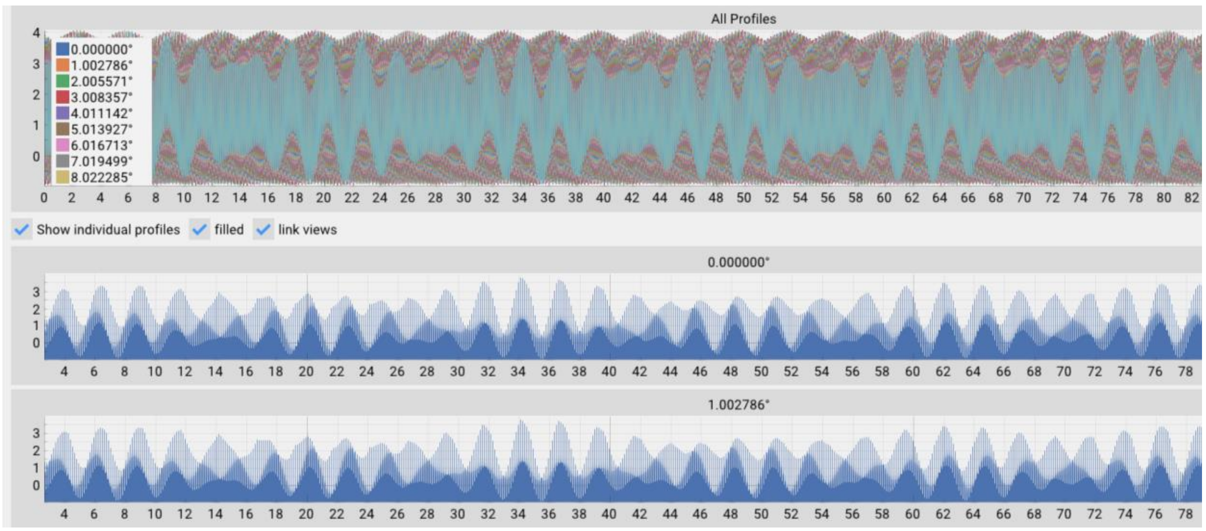


Fig. 9: *Exemplary axial surface profile plots with results from a simulated micro topography (at different profiles: all profiles combined, 0 degrees, and 1.002786 degrees).*

resources. From this microscopic topography data, we want to derive the resulting perturbation of the visual reflection effects.

The first step is to create a height-field from those discrete profiles. Basically, all discrete profiles are copied into one big chunk of data. This height field is then uploaded to the GPU, as real floating-point data, so that no information is lost. Older GPU formats used to only allow image data with clamped values in the range [0...1], which most of the time also involved implicit bit-reduction. So now, with the evolution of technology, no error-prone scaling of values must happen in the shader computations. For direct visualization of that height field, we still create a clamped version of it (Fig. 10). From that we can interpolate height offsets for each angular position on the workpiece surface (unwrapped to tangent space). In Fig. 10 (bottom) this map is directly mapped as red color onto the workpiece. One can already see a slight interference pattern from that, which is not easily visible in the 2D map.

Next step is the calculation of a normal map: From the local surface angles along and between the discrete micro profiles in the height map, we create 3D normal vectors. That normal map does not actually exist, but the vectors are calculated on the fly from the interpolated height filed positions, where needed (Fig. 11). They are literally derived in tangent space, as partial derivatives in axial and angular direction, from the height field. This means that we have no upfront limitation regarding the final normal map resolution like we might have with pre-calculated textures. Of course, the resolution of the original micro profile data is still a limiting factor – but even with a low count of discrete profiles we could apply different interpolation and filtering strategies to improve quality as compared to a pre-computed low resolution normal map. In Fig. 12 we can see the difference that an interpolated (right) vs non-interpolated (left) height field makes: There are much more reflection details visible in the interpolated version.

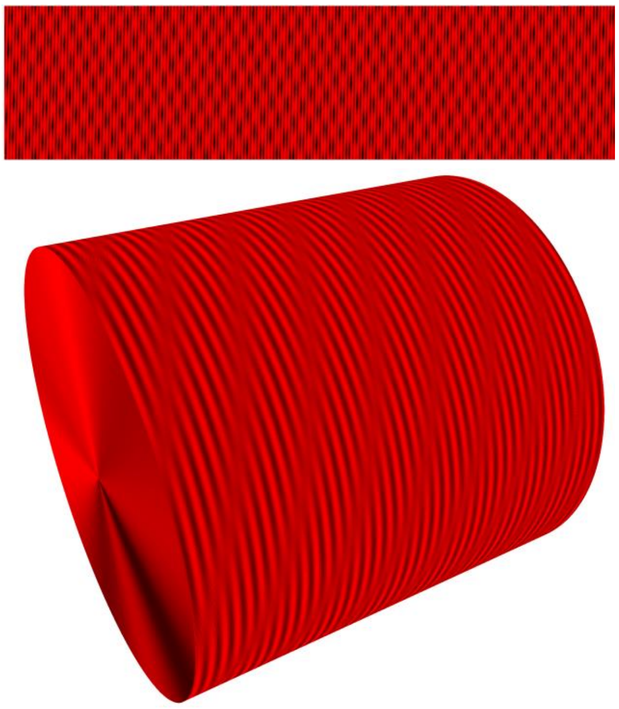


Fig. 12: Axial profiles combined into a 2D topography map (top), mapped onto the workpiece (bottom).

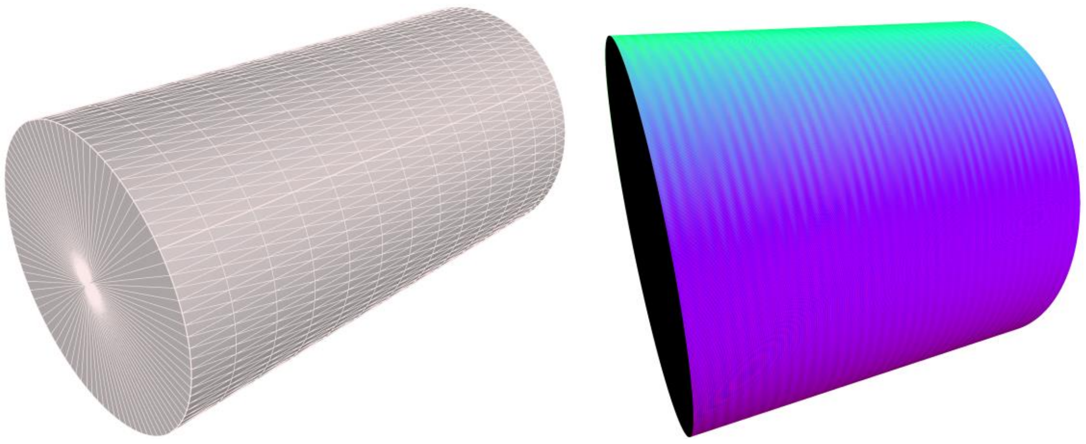

,

Fig. 10: Low resolution workpiece (left), with per fragment derived normal vectors, visualized as RGB colors (right).

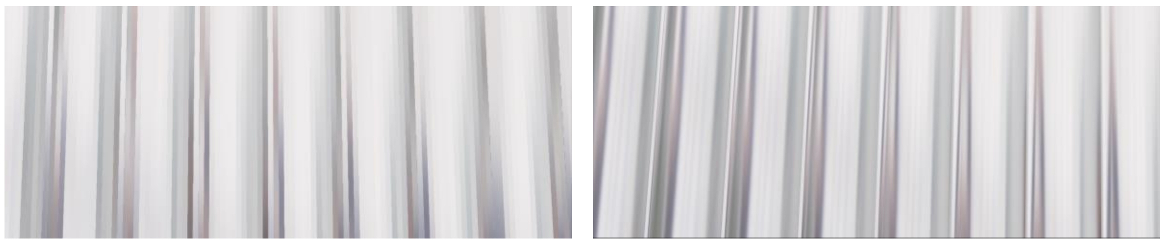


Fig. 11: Close-up of reflection details in grooves on the workpiece surface without interpolated surface topography (left), with interpolated surface topography (right).

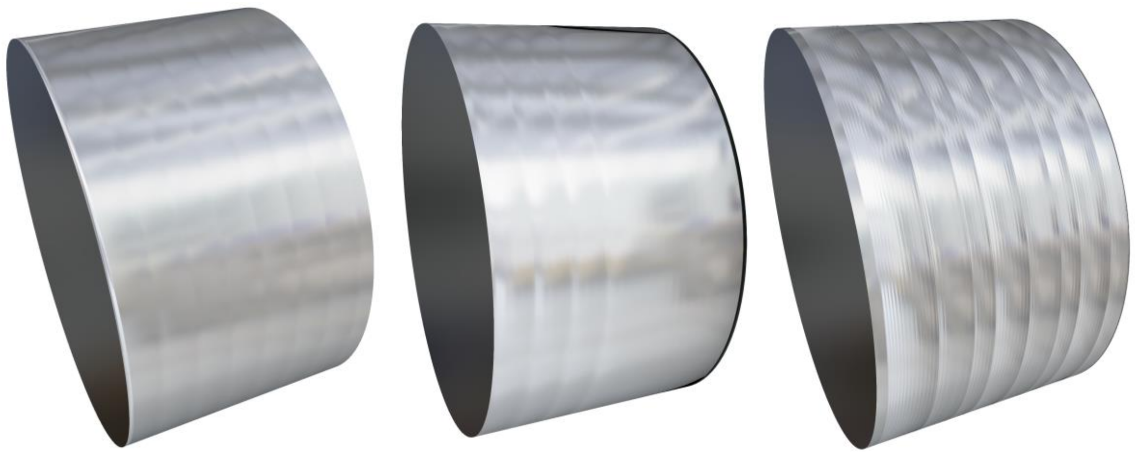


Fig. 14: Visualization of synthetic vibration patterns as generated in our prototype application. Increasing number of amplitudes and vibrations from left to right.

For each fragment, we want to determine the corresponding surface properties. More precisely, we want to perturbate the original workpiece surface normal with the corresponding normal that we derived from the height map, in tangent space. Because those two normal vectors are defined in different spaces, we need to apply some transformations so that we can relate them. The TBN (Tangent Bitangent Normal) matrix defines that transformation and is strictly speaking different for each single surface point. We can construct the tangents from the input geometry, as we assume a cylindrical workpiece. For the more generic case, the input geometry would also need to supply tangent vectors. From that we can combine the TBN matrix and modulate the original workpiece surface normal accordingly. It will then be perturbed according to the surface topography described by the axial micro profiles.

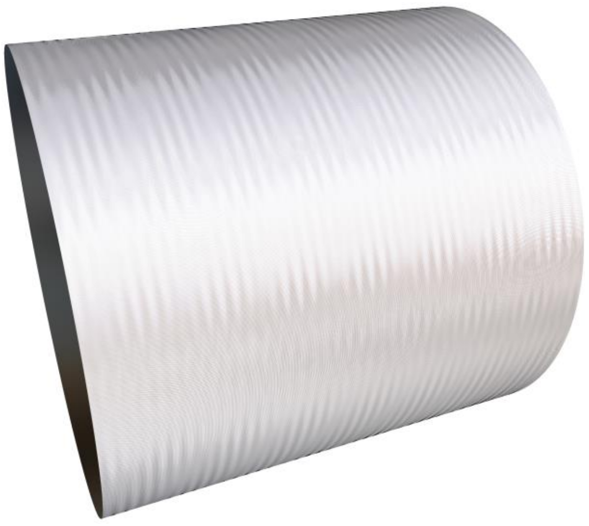


Fig. 13: Final visualization of turning process simulation with vibrations. Workpiece dimensions 100x100mm, 1400rpm, feed rate 0.14 mm/rev, tool radius 0.8mm, Vibrations at 91.93Hz amplitude 0.5µm, and 117.83Hz with amplitude 0.5µm (Effects have been exaggerated to be visible in the still image).

This normal then is used, as described before, in the usual transform and lighting calculations for the base color, and especially for the environmental texture map lookup which creates the visual reflection effects).

We have used this method with synthetical data, which can be generated directly in our prototype application (Fig. 14), as well as from some more sophisticated simulation results from one of our partners (Fig. 13).

6 REAL TIME RENDERING INFRASTRUCTURE

For the visualization parts of our prototype application, we are using standard C++ and the OpenGL API directly, as mentioned before. For the application user interface, we utilize some small libraries: 'Dear ImGui' [ImGui] for the user interface, 'ImPlot' [ImPlot] for graphs, e.g., of the micro profiles, and of course the cutting simulation kernel, which provides the triangle model of the visualized workpiece.

The prototype implementation uses plain OpenGL v4 with GLSL shaders. We utilize Vertex Shaders, Fragment Shaders, and (mainly for debugging purposes) also Geometry Shaders.

OpenGL v4 also offers Compute Shaders in case we may need them in the future. Compute Shaders basically allow to execute shader programs on the GPU without also throwing geometry at it which used to be the only way to execute shader programs with former OpenGL versions. With OpenGL v4 data formats used by shaders can be selected much more arbitrary and are no longer limited to geometry and image data types only. This not only applies to Compute Shaders but to all Shader stages. Thus, it is no longer necessary to dress up data as textures which removes most burdens that used to be connected to using graphics shaders for 'general purpose' (GPGPU) applications. At the same time, this could render CUDA or OpenCL obsolete for a lot of types of applications, especially when visualization is involved that uses OpenGL anyway. In contrast to CUDA or OpenCL, OpenGL is much more compatible across devices and platforms, is not vendor-specific (at least in theory). It also does not introduce the huge toolset and runtime-library overhead that especially CUDA needs. In fact, besides a capable graphics driver, OpenGL needs no additional libraries or tools at all.

Lately we started to develop some custom utility to make the shader development much faster: Our prototype application offers to (re)load shaders at runtime from files and at the same time automatically generates UI elements for the detected shader variables. This means that, as developers, we can exchange whole shaders, modify and add available parameters, and change their values on the fly without even stopping the application. Not to mention changing application code, re-building, re-starting and re-simulating the stock model before seeing results of changed visualization parameters. Even the virtual camera can stay where it is, and changes are immediately visible. A major use case for those on-the-fly change of variables are switches which toggle different code paths in a shader program. This is, more or less, the only way to debug GPU shaders efficiently. The ability to adjust all this on the fly while visually inspecting the surface visualization safes hours of development time.

The development currently is done on different hardware devices, specifically on NVIDIA Quadro P1000 and NVIDIA GFX 1060 graphics boards. Also, an Intel UHD 630 is used, but it caused some unexpected results because it does not always behave as defined, depending on driver versions (a well-known vendor-specific problem among OpenGL developers).

Up to now we have not seen performance bottlenecks coming from those rather settled GPUs, although there is probably a lot of room for performance optimization in our prototypes.

7 CONCLUSION

We have shown different approaches to real time rendering of workpiece surface properties. Starting from plausible, manually controlled material visualization for a grinding process, we mixed in some more simulation-result controlled properties to differentiate surface properties by broaching tool teeth. In the last examples, we presented how we apply actual microscopic surface topography simulation results to our mesoscopic workpiece visualization.

Up to now we focused on involving actual simulation results into our visualization approach. We might add more realistic looking, physically based material models to our visualization in the future, i.e., the Cook-Torrance reflectance model seems to be a good fit for metallic shading. This would mean that the base color to which we add our reflection effect would also look more metallic, even without the reflection effects.

We showed that the combination of mesoscopic simulation results for a workpiece geometry with information about the microscopic surface properties can be utilized to visualize resulting surface quality.

8 ACKNOWLEDGMENTS

This research was partially funded from the European Union’s Horizon 2020 Research and Innovation. Programme under the project InterQ (grant agreement No. 958357), and it is an initiative of the Factories-of-the-Future (FoF) Public Private Partnership.

9 REFERENCES

Paper in a journal:

[Bilalis 2009] Bilalis, N., Petousis, M., and Antoniadis, A. “Model for Surface-Roughness Parameters Determination in a Virtual Machine Shop Environment.” The International Journal of Advanced Manufacturing Technology, The International Journal of Advanced Manufacturing Technology, 2009, 40 (11): 1137–1147.

[Blinn 1976] Blinn, J. F. and Newell, M. E., Texture and reflection in computer generated images. Communications of the ACM, 1976, Volume 19, Issue 10, pp 542-547

[Blinn 1978] Blinn, J. F., Simulation of wrinkled surfaces. ACM SIGGRAPH Computer Graphics, 1978, Volume 12, Issue 3, pp 286-292

[Brecher 2017] Brecher, C., Wellmann, F., and Epple, A. “Quality-Predictive CAM Simulation for NC Milling.” Procedia Manufacturing, Procedia Manufacturing, 2017, 11: 1519–1527.

[Gustavson 2022] Gustavson, S. and McEwan, I., Tiling simplex noise and flow noise in two and three dimensions, Journal of Computer Graphics Techniques, 2022, Volume 11, Issue 1, pp 17-33, ISSN 2331-7418

[Klimant 2014] Klimant, P., Witt, M., and Kuhl, M. “CAD Kernel Based Simulation of Milling Processes.” Procedia CIRP, Procedia CIRP, 2014, 17: 710–715.

[Liu 2005] Liu, N., Loftus, M., and Whitten, A. Surface finish visualisation in high speed, ball nose milling applications. International Journal of Machine Tools and Manufacture, 2005,45(10), 1152–1161

[Perlin 1985] Perlin, K., An image synthesizer. ACM Siggraph Computer Graphics, 1985, Volume 19, Issue 3, pp 287-296

[Phong 1975] Phong, B., Illumination for computer generated pictures. Communications of the ACM, 1975, Volume 18, Issue 6, pp 311-317

[Wang 2016] Wang P, Zhang S, Li Z, Li J. Tool path planning and milling surface simulation for vehicle rear bumper mold. Advances in Mechanical Engineering. 2016;8(3). doi:10.1177/1687814016641569

Paper in proceedings:

[Benouamer 1997] Benouamer, M. O. and Michelucci, D., “Bridging the gap between CSG and Brep via a triple ray representation.” in Proceedings of the fourth ACM symposium on Solid modeling and applications - SMA’97, 1997.

[Blinn 1977] Blinn, J., Models of light reflection for computer synthesized pictures. In: Proceedings of the 4th annual conference on Computer graphics and Interactive techniques, July 20-22, 1977, San Jose, California: ACM NY, pp. 192-198, ISBN 9781450373555

References to online resources:

[ImGui] Dear ImGui graphical user interface library, https://github.com/ocornut/imgui

[ImPlot] ImPlot plotting library for ImGui, https://github.com/epezent/implot