REAL TIME 3D SIMULATION TOOL FOR NC SHEET METAL CUTTING AND PUNCHING PROCESSES

Aitor Moreno
Vicomtech
email: amoreno@vicomtech.org

Álvaro Segura
Vicomtech
email: asegura@vicomtech.org

Harbil Arregui
Vicomtech
email: harregui@vicomtech.org

Álvaro Ruíz de Infante
Lantek Investigación y Desarrollo S.L.
email: A.RuizDeInfante@lantek.es

Natxo Canto
Lantek Sheet Metal Solutions
email: N.Canto@lantek.es

May 15, 2012

KEYWORDS
Computer Integrated Manufacturing and Engineering (CIME), Industrial processes, Optimization, Interactive simulation, Real time simulation

ABSTRACT

In this paper we present a sheet metal Numerical Control (NC) simulation tool for cutting and punching processes, considering the internal representation of the sheet metal as a 2D complex polygon. The nature of the involved processes in the cutting and punching operations supports the utilisation of Boolean Operations between 2D polygons with fully established geometrical methods. However, straightforward utilisation of such Boolean Operations to support the material removal process leads to slow simulation times, since complexity of the sheet increases continuously. Some optimizations have been introduced to outperform the simulation times, such as the spatial subdivision and optimized methods to generate directly the swept area in arc movements. Results show that those optimizations are significant and have a direct impact in the simulation performance.

INTRODUCTION

In the machining industry, technological advances have led to increased productivity and business efficiency. Generally, these advances have been higher in mechanical technology, obtaining more efficient machinery, faster and more versatile, leading to better economical results. The management and optimization of resources is one of the most interesting challenges, since a small improvement (in time or resource consumption) results in a real savings in the processes that will produce thousands of parts. In the field of sheet metal cutting it is even more crucial because the very nature of the cutting process will generate wasted material, having to be returned to the melting industry for recycling.
Therefore, the simulation tools help to test and check the programs in the design phase, being verified all the necessary times before they are actually run in the actual machine. After a series of tests, including the optimization modifications, the program can reach an optimized state, good enough to be transferred to production. Any not fully tested Numerical Control (NC) program can cause or increase the risks and compromise the machinery, provoking partial breakages of parts, collisions between different machine parts, the sheet metal and the tool.

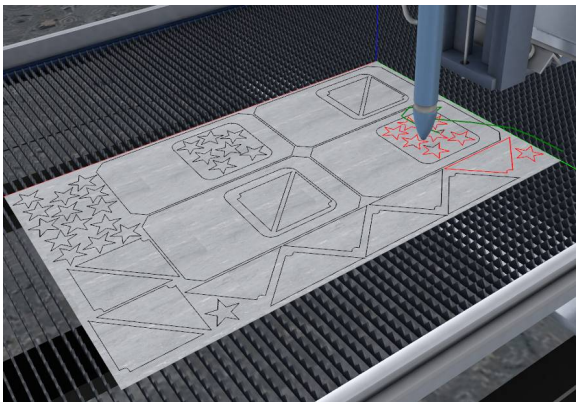

Figure 1: Sheet metal cutting simulation example.

This work presents a NC simulation tool (see Fig. 1) specifically aimed for the virtual representation of the metal sheet main machining processes (see Fig. 2), like cutting and punching. The first part of this work introduces the technological advances in the field, being the basis for the implemented simulation system, presented in the corresponding section. The performance improvements of the proposed architecture are addressed, followed by the conclusions and some future guidelines.

RELATED WORK

The NC machining simulation using Computer Graphics techniques is a widely extended research topic, where

Figure 2: Real sheet metal cutting and punching operations.

the main issue is related to the representation of the dynamic parts of the simulation (in this case, the sheet metal is considered dynamic, as its geometry changes over time). Some traditional approaches do not store the geometrical information during the simulation, but they simply modify the drawing screen using an image-based approach.

Some techniques store the intermediate results in the computer's memory, having an internal 3D geometric representation of the object that is changed continuously during the simulation process. With these methods, a permanent representation is always available and provides free camera movement around the object, better geometric accuracy control, geometric based collision detection, etc.

Van Hook (Hook 1986) used an extended Z-buffer data structure (called a Dexel structure) for the graphical verification. In his work, a scan method to convert surface data into his Dexel (depth element) structure was presented. The Z values for the nearest and the farthest surface at each Dexel are stored in such depth elements. This technique has been extended by several authors (Zhu and Lee 2004).

Other representation methods in the Computer Graphics field are fundamentally geometric like *i)* Boundary Representation (B-Rep) *ii)* Constructive Solid Geometry (CSG), and *iii)* Hierarchical Space Decomposition (HSP).

Although B-Rep is the most used method for solid modelling in modern CAD systems, its straightforward use for machining simulation is not convenient due to the long time required for the dynamic simulations (Spence and Li 2001). A similar problem occurs with CSG representation, with computational costs of order $O(n^2)$, where n is the number of primitives (Stewart et al. 2003) being computationally expensive. A modern implementation of CSG representation through BSP (Binary Space Partitioning) trees has been ported to Javascript and to the Web with a great but non real time performance (Wallace 2012).

To cope with the complexity of the problem and the long time required in these approaches, the approximation of the exact geometry, and especially the partitioning of the object in suitable regions has been proposed by several authors (Stewart et al. 2003).

The most classic technique for volume partitioning is the voxel representation (classical octree, extended octree (Brunet and Navazo 1990), SP-Octree (Cano 2002)) that combines the space partitioning, solid representation and boolean operation support in a single definition. The sheet to be manufactured can be approximated by a very thin extruded plane, given that the machining program is limited to 2D movement over the sheet. This sheet representation provides a direct way to perform boolean operations between the moving tools and the planar sheet. The 3D boolean operation is simplified in a single 2D boolean operation between two 2D complex polygons, that is a well reviewed research topics (Vatti 1992, Preparata and Shamos 1985).

In this work, we present how an efficient and optimized sheet metal machining simulator has been designed and developed, with an internal geometrical core based on complex polygon boolean operations to support the material removal processes.

METHODOLOGY

This work is aimed to develop a simulation system for cutting and punching of sheet metal through a NC controller. The simulator has been developed as a prototype simulation software module and it is focused to obtain consistent geometric results and high graphics quality for the following sheet machining processes (see Fig. 3):

- Sheet metal cutting processes using laser, plasma, oxy-fuel or water jet.
- Punching processes with tools defined with basic or complex shapes.

By means of simulation techniques, this work tries to emulate the behavior of the machine tool to the computer. The simulation system takes as input a starting NC program (normally, G-code dialect) translated to

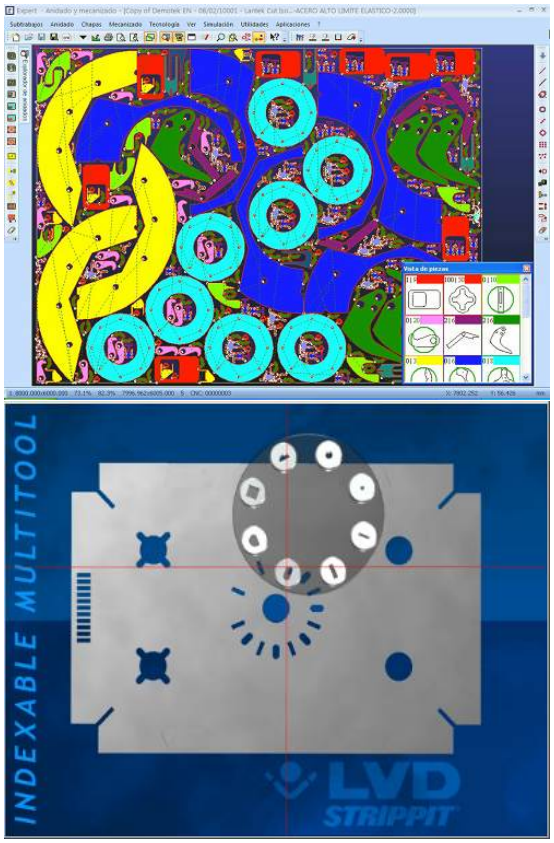

Figure 3: Nesting example for sheet metal machining (Lantek 2012) and a Virtual Punching 2D visualization (LVDGroup 2012).

a common and simplified XML format, listing all the movements that the machine will perform during the cutting or punching operation. The operation mechanisms differ between the cutting and punching process, but essentially, they are based on the removal of material from the sheet, so the internal module for such operations has been designed to be generic for such operations. The main significant difference is that in the cutting process the removal process is continuous while the torch is powered on, whereas in the punching process, the removal process is instantaneous when the punch is triggered.
In the following subsections, the system architecture and the main modules will be described.

System Architecture

The simulator is structured as a multi-layered architecture, each one encapsulating different methods and techniques. The low level layer involves the geometry calculations (Geometric Kernel) with the management of the boolean operations between 2D polygons as its main responsibility. The Clipper library (Johnson 2012) provides the functionality to calculate Boolean Operations between 2D polygons, as the basis for subtraction between a complex polygon representing the sheet and another polygon representing the sweep of a moving tool in a given period of time (either cutting or punching). This low level layer (green box in Fig. 4) provides the access to the graphic system, responsible for rendering the simulation results in the screen. In this work, we have chosen OpenSceneGraph (OpenSceneGraph 2012) as the graphics subsystem, which is used to draw on screen the result of the Boolean operations, the 3D machine models and all the virtual elements in the scene.
The middle layer of the architecture (red box in Fig. 4 provides a conceptual representation of the entities related to the NC machine domain, like Part, Tool, Machine and Axis, including all animation of all the moving elements. Also, in this layer the geometric sweep volume of the moving tools are calculated and passed to the lower level for the actual boolean operation with the sheet metal representation.
Over the cutting and animation layer, we have added the user oriented layer (blue box in Fig. 4), including the graphical interface of the prototype application and management of the XML files:

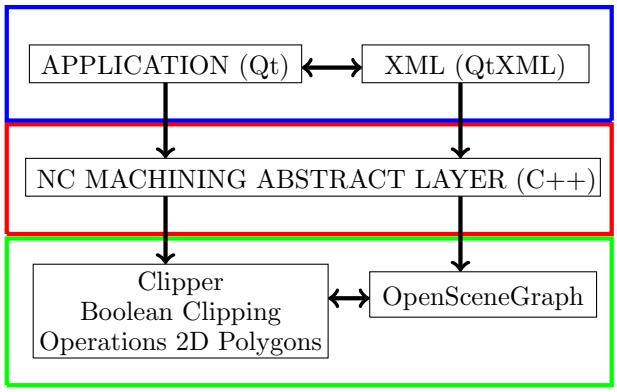

Figure 4: Modular Architecture

- Provide the user interface, including multi-language interface options.
- Implementation of the different navigation methods in the 3D virtual world, using the mouse.
- Manage and display the real NC instructions (highlighting the currently executing instruction) that will be loaded into the simulator through the intermediate and abstract XML representation.
- Manage and display the visualization options of the 3D objects corresponding to the individual parts of the cutting or punching machine.

Both the user interface and the XML parsing functionality have been implemented through the Qt library (blue layer in Fig. 4).

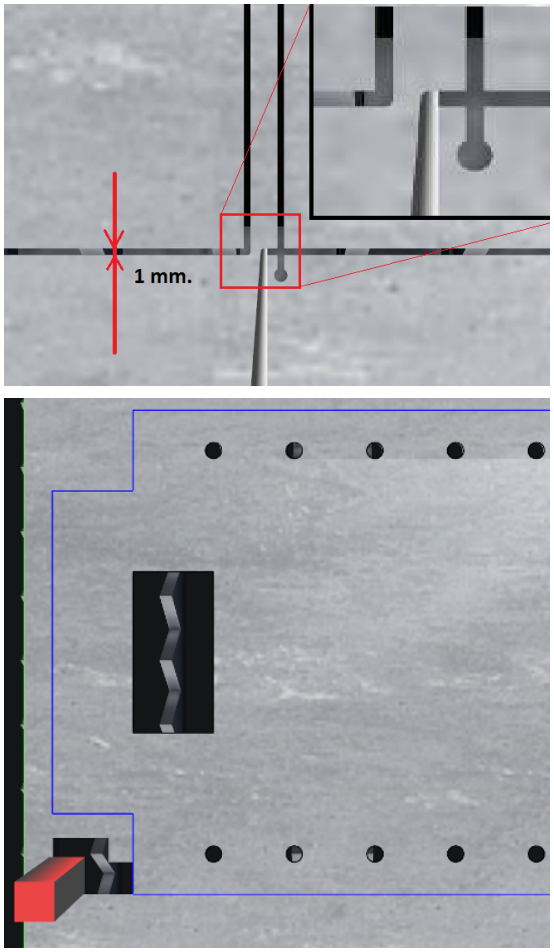

Figure 5: Close up 2D sheet metal machining and punching example.

Listing 1: Pseudo code for the subtraction methods for cutting or punching movements

```
OpBoolCut (in-out Obj sheet, in Mov m)
{
  Obj sweep = GenerateSweep (m)
  BoolOp res = PolyClip2D (sheet, sweep)
  sheet = res
}
OpBoolPunch (in-out Obj sheet, in Mov m)
{
  Obj c = getPunchContour (m)
  BoolOp res = PolyClip2D (sheet, c)
  sheet = res
}
```

Overview of the Sheet Metal Cutting Algorithm

The main elements in the system are the metal sheet and the tools that will translate the machining instructions into geometrical operations. As the tools moves on the sheet, the swept area has to be removed from the sheet, thus, after applying multiple subtraction boolean operations, the sheet will be modified, resembling the expected output in the real world.

The foundation of the methodology is a 2D polygon clipping system. The cutting element or torch (laser, plasma, waterjet, etc.) can be modelled as a cylinder of varying radius (see Fig. 5, top). In a time interval, the moving cutter intersecting the sheet metal sweeps a shape than can be represented by a polygonal contour with curves approximated by sequences of linear segments. In each time step, such swept polygon is subtracted from the polygonal representation of the sheet containing all previously removed contours, setting the result as the new sheet. Additionally, from this new updated geometric representation of the sheet, the graphic subsystem must be updated too, so the final rendering on the screen is updated consequently (see Listing 1).

The geometric representation of punch tools is more complex than the torches, since there are circular, square, rectangular or even arbitrary shapes for punching processes. But the simulation process is simpler as these tools are not activated in continuous mode. A *Punch* instruction activates the current active tool, making a hole in the sheet and returning to its original position. This process can be done multiple times, at a very fast pace, providing similar features to the sheet metal cutting processes (see Fig. 5, bottom).

Cutting and Punching Machine Specification

Cutting and punching machines are essentially different, but at the same time, they have similar components and a generic hierarchical structure can be defined.

The hierarchical structure of the 3D model of the cutting machines starts with the *Table*, as the static part of the machine. Over it, the *Bridge* moves in the X axis. The *Torch Support* is mounted over the bridge as the Y axis. The movements in the Z axis of the machine are performed by the *Torch*, fixed to the support structure (See Fig. 7). Although we have developed several 3D model for the different machines we have considered (oxy-fuel, plasma, laser and waterjet), they all follow the same hierarchical definition.

The hierarchical structure of the punching machine is significantly different from the cutting machine. In the punching machine, the sheet is the mobile element, while in the cutting machine the sheet is static. Therefore, the sheet is moved in the X and Y axis, while the punching stations give the punch movements in the Z axis. The implemented prototype of the punching machine has room for up to 20 individual punching tools. The individual punch geometries are loaded from an XML file.

The visualization of the punching machines requires playing with transparencies in the punching station and the bridge, since they hide completely the punching operation (See Fig. 6).

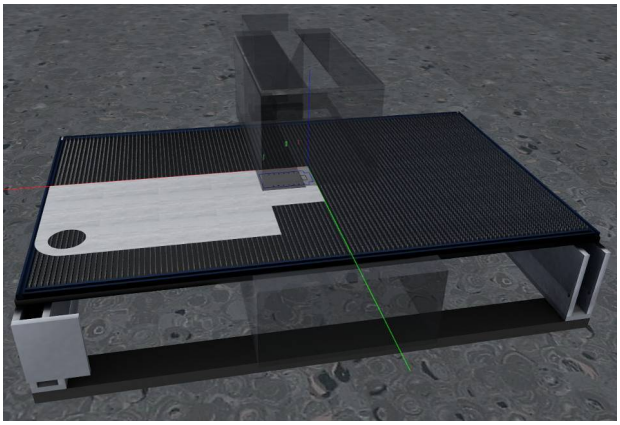

Figure 6: Simplified 3D model for the punching machine with transparent elements to ease the visualization of the punching process.

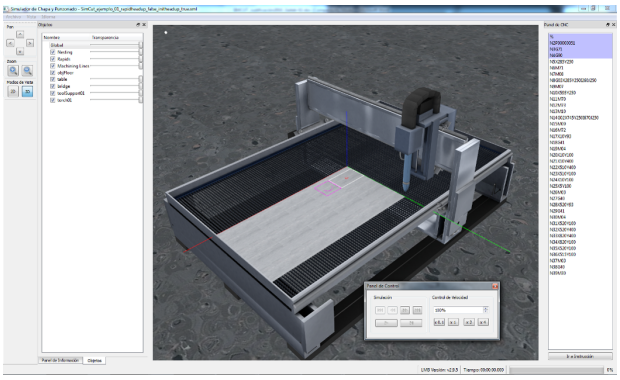

Figure 7: Prototype loaded with a program for cutting a rectangular metal sheet and configured for a single oxy-fuel torch.

User Interface

The user interface displays a simple and clean interface between the user and the simulation functionality. It has been developed using the Qt library, which provides the technology needed to create professional and high-quality graphical interface controls. The application GUI is structured as a set of docking and floating panels, which can be moved freely or docked to any side of the screen (see Fig. 7) with the 3D virtual world always in the central widget.
There are two ways of controlling the virtual simulation. One is by running a continuous simulation, where an animation factor is applied to control de simulation speed. The other is a fast mode, which will run the simulation in background as fast as possible, until the target instruction is reached. Combining both modes, users can go to a specific instruction of the NC code (shown in an independent panel) and from there, begin an animated simulation with the desired speed factor. The rest of the buttons in the Control Panel offer simple VCR functionality: *play and stop*, *play only one instruction*, and go directly to the previous or next instruction, the first or the last instruction of the loaded program.
The GUI provides the visibility options for all the objects in the virtual scene. Any object can toogle its visibility, but not all the elements can modify the transparency, as the lines (machining toolpaths and other helping elements) and the ground model. The camera properties and movements can be set up in another panel, including the toogle button to go to 3D or 2D mode and the Zoom and Pan functionality.
Finally, thanks to the multilingual support from Qt, the interface has been easily translated to several languages, with an easy and portable mechanism to add new languages.

PERFORMANCE OPTIMIZATIONS

As the NC program instructions are known before hand, just when the simulator is started, it is possible to preprocess some information. However, these actions cannot be performed if a full run of the simulation is requested as soon as the simulation is started. In this worst scenario, all the machining instructions must be run in the fastest way, optimizing the resources of the hosting computer. Additionally, the state of the simulator after the final instruction is reached must keep the interactivity of the application for further simulations.
In the following subsections, some applied techniques are presented to try to optimize the overall performance in term of simulation time and other resources consumption such as memory or hard disk.

Direct Contour Generation for Arc Sweeps

The circular movements of the tools were initially implemented by a subdivision into linear piecewise movements. We used a global variable in the system representing the number of subdivisions that a full circle would have. If this value is 32, a full circle movement produces 32 linear movements and similarly, a π radians arc will be decomposed in 16 pieces. The immediate drawback of such subdivision system is that the number of boolean operations grows significantly.
In order to avoid extra boolean operations, we have implemented a direct or native generation method for the arc movements. Using such functionality, the swept contour of a tool is generated directly, and thus, a single boolean operation is performed (see Fig. 8), resulting in a significant performance boost (see Table 1).
The subroutine *AddArcPoints* used in Listing 2, run as *p.AddArcPoints (p1, p2, c, CCW or CW)*, samples a given arc movement going from point *p1* to point *p2*, with point *c* as the center of the arc, in the given *sense*, clockwise or counter clockwise. The resulting sampled points are appended at the end of the *Polygon p*. The

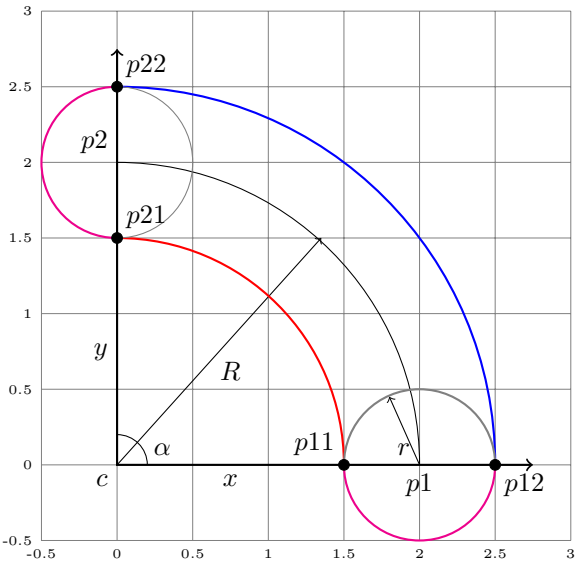

Figure 8: Sweep arc geometric construction for a tool with radius r moving from point *p1* to point *p2* in an arc movement. The sweep contour is constructed by sampling four partial arcs, using the points *p11*, *p12*, *p21* and *p22* as the limit points.

GetCorners subroutine calculates the point coordinates where the individual sectors of the arc are joined.

Listing 2: Pseudo code for the SweepArc function

```
SweepArc (in Point p1, in Point p2,
          in Point c, in float r,
          out Polygon p)
{
  GetCorners (pc11, pc12, pc21, pc22)
  p = Polygon::CreateEmptyPolygon()
  p.AddArcPoints (pc11, p21, c, CCW)
  p.AddArcPoints (pc21, p22, p2, CW)
  p.AddArcPoints (pc22, p12, c, CW)
  p.AddArcPoints (pc12, p11, p1, CW)
}
```

The actual method *SweepArc* is more complex than the pseudo-code shown in Listing 2, as there is a number of cases that should be treated one by one, e.g., when $p1 = p2$ or $r > R$. Additionally, a better consistency with the sense of the arc sampling is desired in order to get geometrically accurate contours for the tool sweep.

Spatial Subdivision

The efficiency of the polygon clipping algorithms depends directly on the total number of contours and points involved in the boolean operation (Leonov 1998):

$$O(n \times \log(n) + k + z \times \log(n)) \quad (1)$$

Table 1: Performance improvement using optimized sweep arc generation. A full simulation was performed with and without the optimization and the number of low level Boolean Operation and the time are presented in the columns *OpBools* and *Time*.

Arc Optimization	Movs	OpBools	Time (s)
No	560	1054	25
Yes	560	774	10

where n is the number of edges (points), z is the number of contours and k is the number of edge intersections.

As the simulation is performed, the working part gets more and more complex and consequently, the number of points and contours grows. In order to limit the number of points and contours that would increase the boolean operation time, a high-level partitioning system is added to the architecture.

This spatial partitioning decomposes the sheet metal into a set of smaller subregions, leading to a high level Boolean Operation pseudo-algorithm (see Listing 3).

The performance effect of the spatial subdivision is limited as an over-subdivided sheet will increase the number of individual Boolean Operations, as any movement will span across multiple regions. The subdivision region is aimed to reduce the complexity of the Boolean Operation (by limiting the number of vertices) but avoiding to increase the mean number of Boolean Operations per movement.

Listing 3: Pseudo code for the Boolean Subtraction between the sheet and the sweep

```
OpBool (in-out Obj part, in Obj sweep)
{
  Set S = SelectRegions (part, sweep)
  for each region R in S }
    BoolOp res = PolyClip2D (R, sweep)
    part.SetRegion (R, res)
  end for
}
```

In our experiments (see Table 2), varying the number of subdivision with the same example, gives a performance peak using the 16×16 subdivision. The Fig. 9 shows the geometric complexity of the metal sheet after the simulation program is completely executed. Approximately, each full circle contains around 100 vertices, 50 for the internal circle and 50 for the external circle. The whole geometric sheet contains around 1 million vertices, and due to the spatial subdivision (16×16 in this case), the rendering is performed in real time using the OpenSceneGraph's internal polygon tesselator (OpenGL's GLUtesselator methods).

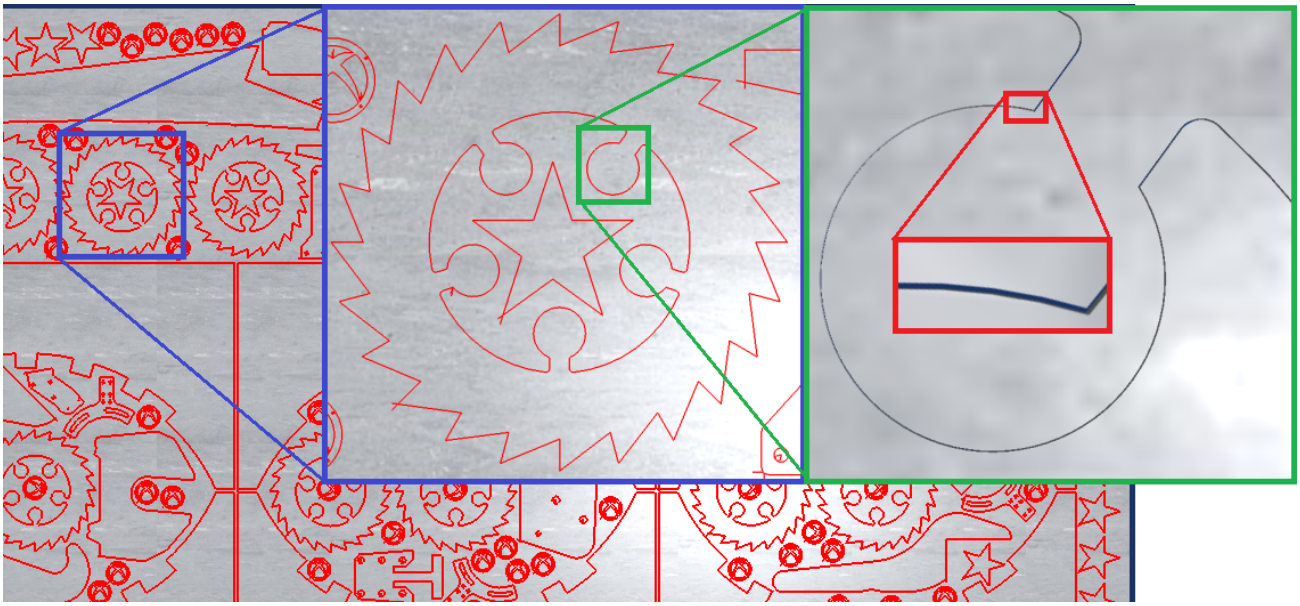

Figure 9: A complex cutting example with more than 12000 movements, combining linear and arc movements. The simulation time is about 90 seconds on an medium range PC.

Table 2: Spatial Subdivision Performance, with the direct sweep arc generation applied, varying the spatial subdivision. The number of low level Boolean Operation and the time are presented in the columns *OpBools* and *T*.

Model	Movs	Subdivision	OpBool	T (s.)
Simple	834	1×1	774	10
		4×4	1008	5
		16×16	2700	6
		32×32	6556	10
Complex	12371	1×1	11037	2100
		4×4	12481	195
		16×16	19533	87
		32×32	36910	130

Saving intermediate states

The VCR functionality enables users to select any desired instruction in the NC program. If the user wants to jump to a non executed instruction, the simulator must run the program till that point in the highest possible speed. But in the opposite way, the simulator should be able to load already calculated simulation points. Unfortunately, in programs with more than 8000 instructions, it is not convenient to save all the intermediate states into memory or even in temporary files in the hard drive, since it would take a lot of resources of the hosting machine.

So, the implemented solution saves a limited number of instructions, acting as keyframes of the simulation. Therefore, when users need to go a previous instruction or, generally speaking, an already calculated position in the CNC program, the simulator will load the previous saved state (the keyframe) and run a small and silent simulation from that point to the target instruction. Of course, the overall performance impact is a matter of balancing the number of keyframes, since a very high number of keyframes will reduce the mean waiting time (the silent simulation step) but will increase the resources usage. With a low number of keyframes, the simulator will use less resources, but the waiting time could interfere with the usability of the simulator, as this seek functionality is widely used to review certain parts of the machining process.

Hardware configuration

The tests and numerical analysis for this Section have been conducted using an Intel Quad Core Q9400 processor, 4GB of RAM and a GeForce GTX 285, Windows 7 PRO 64 bits (Service Pack 1) with the latest stable graphics drivers.

CONCLUSIONS AND FUTURE WORK

In this work we have presented a simulator for the sheet metal cutting and punching processes. Due to the characteristics of the sheet, a representation based on 2D complex polygons has been used to represent the metal sheet. All the operations of the programs have been transformed to internal boolean operations between the

sheet and the sweep of moving cylinder (cutting machines) or a complex polygon (for punching machines). As the efficiency of the boolean operations decay with the number of points and polygons, we have introduced several mechanism to optimize the overall result of the simulation. To limit the number of points and polygons, the spatial partition system has been used, checking that it can not be increased to arbitrary numbers, since it will provoke a huge explosion in the number of boolean operations. A balanced spatial partition is required to get the best performance. In the future, a more detailed analysis of such conditions should be evaluated.
The optimizations for the generation of the sweeps in arc movements have provided a boost to the effiency, too. When applied, the number of boolean operations decreases, replacing several operations by just a single one with significant performance improvement. However, the sweep generation can be further improved as the current method samples the arcs independently of the radius of the arc movement and the tool. For example, an optimized version of the function would sample the inner arcs with fewer points than the exterior arcs.
As the sheet metal cutting simulator is intended to be utilised by experts in their field, a fully usable prototype has been developed, providing a GUI to interact with the simulation, displaying all the important information about the NC code, and other features explained in previous sections.

ACKNOWLEDGEMENTS

We thank the Basque Government Industry Department for the financial help received under the GAITEK research.

REFERENCES

Brunet P. and Navazo I., 1990. *Solid representation and operation using extended octrees.* In *ACM Transactions on Graphics 9, 2.* 170–197.

Cano P., 2002. *Representation of polyhedral objects using sp-octrees.* In *Journal of WSCG 10, 1.* 95–101.

Hook V., 1986. *Real Time shaded NC Milling Display.* In *SIGGraph86, Volume 20, Number 4.* 15–20.

Johnson A., 2012. *Clipper - an open source freeware polygon clipping library.* URL `http://www.angusj.com/delphi/clipper.php`.

Lantek, 2012. *Lantek Expert CAD / CAM system.* URL `http://www.lanteksms.com/`.

Leonov M.V., 1998. *Implementation of boolean operations on sets of polygons in the plane.*

LVDGroup, 2012. *LVD - Sheet Metalworking tools.* URL `http://www.lvdgroup.com/`.

OpenSceneGraph, 2012. *Open source 3D Graphics API over OpenGL.* URL `http://www.openscenegraph.org/`.

Preparata F.P. and Shamos M.I., 1985. *Geometry: An Introduction.* Springer-Verlag. ISBN 0-3879-6131-3.

Spence A.D. and Li Z., 2001. *Parallel processing for 2-1/2D machining simulation.* In *Proceedings of the sixth ACM symposium on Solid modeling and applications.* ACM, SMA '01. ISBN 1-58113-366-9, 140–148.

Stewart N.; Leach G.; and John S., 2003. *Improved CSG Rendering using Overlap Graph Subtraction Sequences.* In *International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia.* 47–53.

Vatti B.R., 1992. *A Generic Solution to Polygon Clipping. Communications of the ACM*, 35(7), 56–63.

Wallace E., 2012. *Constructive solid geometry on meshes using BSP trees in JavaScript.* URL `http://evanw.github.com/csg.js`.

Zhu W. and Lee Y., 2004. *Product prototyping and manufacturing planning with 5-DOF haptic sculpting and dexel volume updating.* In *Haptic Interfaces for Virtual Environment and Teleoperator Systems.* 98–105.