A Hybrid Pipeline for Large-Scale PCB Modeling

Qi Zheng Yanning Xu* Lu Wang
School of Software, ShanDong University
JiNan, China
xyn@sdu.edu.cn

Abstract

Since a PCB consists of fixed types of geometric elements, its 3D model can be easily generated from the 2D design. However, a single PCB may contain millions of elements to model, traditional 3D elements generation methods often suffer from high computational and storage cost. To address these challenges, we propose a hybrid pipeline that integrates procedural and non-procedural modeling for different types of elements on modern GPUs. In which, a procedural modeling is performed on those elements with regular shapes, including trail, arcs and circles, by using a GPU-friendly dynamic algorithm to achieve both storage-free modeling and accelerated rendering. Unlike this, the area elements with complex geometric shapes are modeled in a non-procedural way, and a parallel Delaunay-based triangulation is designed for 2D polygonal regions with holes. Compared with the general EDA software Allegro17.3, our method achieves comparable modeling and rendering quality while requiring only 20% of the storage and delivering a 50× speedup. Our pipeline enables fast modeling and real-time rendering of PCBs containing 800,000 elements, achieving a frame rate exceeding 65 FPS which breaks through the modeling limitation of Allegro.

Keywords: PCB, GPU Acceleration, Parallel Modeling, Delaunay Triangulation

1. Introduction

As a critical technology in computer graphics, 3D modeling has been widely applied across various fields, such as industrial design and manufacturing, medical imaging, and film production. In PCB (Printed Circuit Board) design, 3D modeling technology is also becoming an essential tool for design, simulation and manufacturing. By constructing and rendering 3D PCB models, designers can directly observe the stacking structure of board layers, the component layout, and the spatial relationships between traces. Moreover, 3D PCB models can also be used to evaluate electromagnetic compatibility (EMC) and thermal management performance, providing greater assurance for the design process[18, 26].

Currently, PCB data standards such as ODB++ (Sec.2.1) are mostly based on 2D descriptions. However, softwares such as Allegro[1] have already started to emphasize PCB 3D functionality. We refer to the PCB 3D modeling and rendering process of Allegro17.3 as the standard pipeline. These tools usually adopt traditional modeling methods and generic geometric kernels[11] to handle geometric computing tasks and using the vertex shader for 3D rendering. However, PCB data contains a large number of repeated geometric entities (Fig.1), which are widely distributed. Traditional modeling algorithms and rendering pipelines often encounter performance bottlenecks when dealing with PCB scene data that is growing rapidly in scale. This results in the standard pipeline being unable to effectively complete the tasks of rapid modeling and real-time rendering of large-scale PCBs. We need to explore novel approaches.

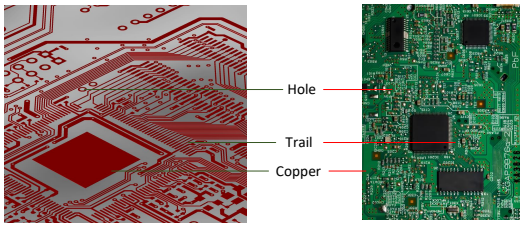


Figure 1. A PCB contains a large number of geometric elements, which can be categorized into three types based on their engineering attributes: copper, trail and hole.

In recent years, the rapid advancement of GPU [12] has facilitated the adoption of various parallel algorithms in the field of geometric modeling. The geometric characteristics of PCBs are inherently suitable for parallel computing, making GPU highly promising for applications in this field. An important observation is that more than 80 percent of elements of PCBs are composed of regular shapes such as holes and trails, which can be realized through procedural modeling. On the other hand, copper areas of PCBs which are represented with both irregular contour and holes are

*Corresponding author: xyn@sdu.edu.cn

difficult to model with rules, and we explore parallel triangulation strategies to improve 3D modeling efficiency. To this end, we propose a hybrid pipeline for PCB modeling and rendering, which integrates specialized data structures and parallel algorithms tailored for PCB modeling. We claim the following contributions:

1. We propose a hybrid pipeline capable of modeling and rendering large-scale scenes, which is applied to PCB engineering.

2. We classified PCB elements and design various GPU-based modeling algorithms to achieve procedural 3D modeling for different types of simple geometric shapes.

3. We improve the existing GPU-accelerated Delaunay Triangulation method to enable triangulation of 2D polygonal regions with holes, and to generate corresponding 3D meshes to better adapt to PCB Modeling.

Our hybrid pipeline demonstrates exceptional adaptability not only in the PCB domain but also in efficiently handling modeling and rendering tasks in other analogous scenarios, such as grass, hair, terrain, medical vascular maps, and brain tractograms. The various modeling strategies of the hybrid pipeline can easily model both regular and complex geometries in the scenarios. Finally, we compare the performance of the standard pipeline(Allegro17.3) and hybrid pipeline. The results highlight the significant advantages of the hybrid pipeline in terms of modeling efficiency and rendering time, underscoring its potential and performance improvements in complex scenarios.

To facilitate reproducibility and further research, we make the source code and part of the datasets publicly available at https://github.com/htl309/PCBBuilder.

2. Related Work

2.1. CAD Geometric Representation

The geometric representation of 3D models is a fundamental topic in computer graphics. Geometric representations are generally categorized into two types: implicit and explicit representations. Common implicit representations include Signed Distance Functions (SDF) and Constructive Solid Geometry (CSG), while explicit representations encompass meshes, voxels, point clouds and others. Each type of geometric representation has its own advantages and limitations. In 1980, Requicha and Aristides G. conducted a comprehensive study summarizing the strengths and weaknesses of various geometric representations[23]. After conducting a thorough investigation, our work adopts the BRep method for geometric representation.

Boundary Representation (BRep) is a commonly used method for precise 3D geometry representation. It combines lightweight parametric curves and surfaces with topological information, linking geometric entities together to describe manifolds[28, 23]. In industrial fields, BRep has widespread applications, particularly in Computer-Aided Design (CAD), Computer-Aided Engineering (CAE), and related areas. ISO 10303, commonly known as the STEP standard, which supports BRep geometry representation, is widely adopted as a guideline for model data exchange. The structured nature of BRep enables STEP files to not only represent individual geometric entities but also effectively represent complex assemblies consisting of multiple components.

ODB++[7] is also a commonly used BRep representation in EDA software. Unlike traditional CAD formats, ODB++ is an accurate geometric 2D representation that improves the consistency of PCB data by integrating the multilayer information of the circuit board into a single standardized format. Our work takes the ODB++ 2D standard as input.

2.2. Parallel Modeling

Parallel modeling refers to the technique of accelerating the modeling process by handling multiple computational tasks simultaneously. GPUs are inherently well-suited for parallel modeling due to their architecture designed for massive parallel computation. Equipped with thousands of processing units, GPUs can simultaneously execute thousands of computational tasks, making them ideal for complex geometric calculations and rendering tasks.

The geometry shader[21, 20] is a stage in the graphics pipeline that processes geometric data passed from the vertex shader and can generate new graphical elements such as points, lines, or triangles. Geometry shader allow for complex transformations and operations on geometry data, which is crucial for parallel computation tasks. It's parallel modeling capabilities have been applied in visualization within fields[13, 24].

Nvidia's Turing architecture introduced the mesh shader architecture [15, 8], which starts with an optional task shader. The task shader serves as a preliminary stage for the mesh shader. It controls on the workflow in a high level, performs task classification and culling, and dispatches mesh shaders. The mesh shader is responsible for computing vertices' location and attributes, as well as a primitive index buffer. The mesh shader not only allows parallel processing of units by thread blocks but also supports up to 32 threads within a single thread block to collaboratively handle tasks within a unit. This mechanism makes the mesh shader highly flexible [16, 14].

Since the introduction of the mesh shader architecture, researchers have begun exploring its potential applications[3]. Benjamin Santerre et al. utilized mesh shader for real-time terrain tessellation [25]. Xiong, Ruicheng used mesh shaders to accurately render large-scale NURBS models [31]. The work of Jérémie Schertzer et al. is most similar to ours, as they used mesh shader to

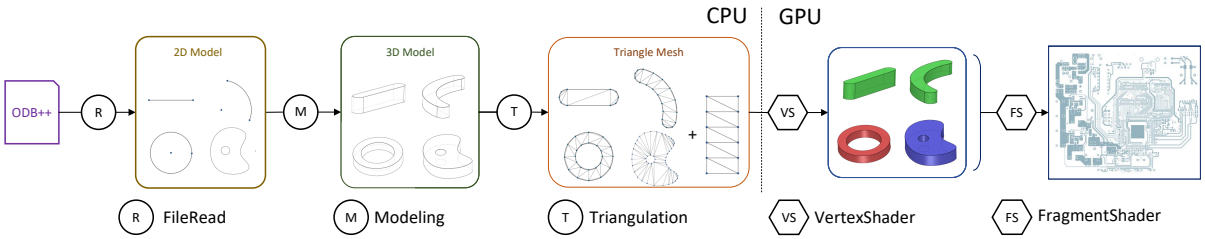


Figure 2. The standard pipeline : First, the PCB data is read from an ODB++ file to construct the 2D data structure. Then, a modeling algorithm transforms it into a 3D representation. Next, discretization and triangulation algorithms generate the mesh, which is ultimately rendered on the screen by the vertex shader.

model and render Massive Brain Tractograms [27], achieving real-time rendering frame rates for brain tractograms consisting of 5.37 billion segments. Tricard Thibault use mesh shader to generate shading intervals for volume rendering[30]. These studies demonstrate the potential of mesh shaders in parallel modeling and rendering of large-scale production 3D.

2.3. Delaunay Triangulation

Triangle mesh is one of the fundamental data structures in computer graphics and geometric processing. Delaunay triangulation is one of the most commonly used algorithms for generating triangle meshes. Its core idea is that for a given set of 2D points, the triangulation generated by Delaunay satisfies the Delaunay criterion, which means that no point lies within the circumcircle of any triangle. Delaunay triangulation can effectively avoid the thin-triangle problem in mesh generation and can be applied to both 2D and 3D spaces [17, 10, 5]. The Constrained Delaunay Triangulation (CDT) algorithm adds constraints in the form of edges to the input point set, requiring the triangulation algorithm to follow these constraints and generate a mesh that maintains topological consistency. In 1987, L. P. Chew implemented a CDT algorithm with a time complexity of $O(\log n)$ using a divide-and-conquer approach [6]. Delaunay refinement algorithms were introduced by Shewchuk, Jonathan Richard, and others, improving the quality of the triangle mesh and enabling it to better fit the requirements of simulations [29].

With the continuous development of hardware, frameworks such as GPU and FPGA have been proposed. Hardware-accelerated Delaunay Triangulation algorithms have gradually come into the spotlight. Yahia S. Elshakhs and others systematically discussed the applications, algorithms, and implementations of Delaunay Triangulation on CPU, GPU, and FPGA [9]. We adopt the work of Qi Meng, Zhenghai Chen, and others [22], who implemented a 2D Constrained Delaunay Triangulation (CDT) algorithm on the GPU, achieving an order of magnitude improvement in the time efficiency of triangulation algorithms compared to the CPU. Furthermore, GPU-accelerated Delaunay refinement algorithms [4] and GPU-accelerated 3D Delaunay Triangulation algorithms [2] are also worth noting. Since we need to perform triangulation on polygons with holes, rather than just sets of points and edges, we extended their work to better adapt the algorithm to our specific requirements.

3. Standard Pipeline

EDA software such as Allegro often relies on 3D CAD kernels to build 3D models for PCB. We observe the facts that PCB primitives to be modeled are relatively simple and the their 3D BRep can be generated from standard ODB++ input by inflation and extrusion. Then most surfaces in their 3D BRep are also rules based and can be triangulated easily. The resulting triangle mesh is rendered through the rasterization pipeline. We refer to this modeling and rendering pipeline, which follows traditional methods, as the standard pipeline (Fig.2).

3.1. Data Structure

As shown in the Tab.1, the PCB contains three types of engineering elements: copper, hole, and trail. Their corresponding 2D and 3D data structures are the area shape/shell, circle seg/shell and 2D/3D trail respectively. In the 2D data structure, an edge is the smallest geometric unit that contains semantic information. The 2D trail is a sequence of edges, which includes both straight line segments and arcs. An area shape is a 2D surface with holes, and it is also represented as a sequence of closed edges. But it includes not only straight lines and arcs, but also NURBS curves. This can be described by the following formulas:

$$2DTrail = \{e_1, e_2, ..., e_n\}, e_i \in \{\text{line , arc}\} \quad (1)$$

$$AreaShape = \{e_1, e_2, ..., e_n\}, e_i \in \{\text{line , arc, nurbs}\} \quad (2)$$

where e_i represents the individual edges of the trail or area shape. Moreover, the edges are all connected end-to-end, forming a continuous sequence (Fig.3).

In the 3D data structure, a shell is the smallest geometric unit. In BRep, a shell is typically represented by a set of

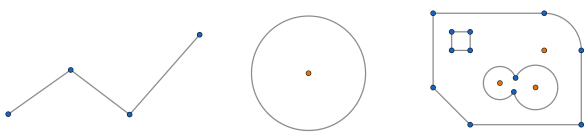


Figure 3. From left to right, 2D trail, circle and area shape.

Table 1. The mapping between the data structures at various stages and the geometric elements in PCB engineering.

PCB	2D DS	3D DS
Copper	Area Shape	Area Shell
Hole	Circle Seg	Circle Shell
Trail	2D Trail (Line Seg, Arc Seg)	3D Trail (LineShell, ArcShell)

boundary surfaces (F_i). This can be described by the following formulas:

$$\begin{aligned} Shell &= \{F_1, F_2, ..., F_n\} \\ F_i &\in \{\text{plane}, \text{cylindrical}, \text{nurbs}\} \end{aligned} \tag{3}$$

Thus, the edges in the Circle and 2D trail are inflated and stretched into a 3D shell, where the connections between edges in the trail are transformed into shell connections.

3.2. Modeling and Rendering

In the standard pipeline, the expansion and stretching methods are typically used for 3D modeling. The modeling of top and bottom faces of trails and circles shells requires expansion (Fig.4). Given a circle with a radius equal to the line width, its center is swept along the line to be expanded. The 2D surface left behind by this sweeping motion is the desired surface. Thus, all trails and circles are transformed into a collection similar to area shapes, which is essentially a 2D surface. Afterward, stretching this surface will yield side faces of the shell.

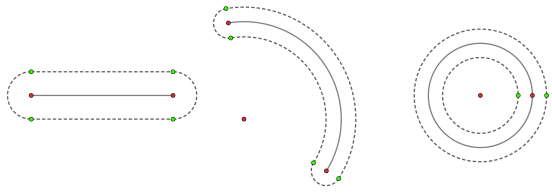


Figure 4. The expansion of Line, Arc, and Circle. The dashed lines represent the contours of the generated faces.

Once the shell is obtained, we can apply discretization algorithms to approximate the arcs, circles, and NURBS edges as sequences of straight lines. Then, the Delaunay triangulation algorithm can be used to generate the triangle meshes for the top and bottom surfaces. Subsequently, the side surface meshes can be obtained using the triangle strip method, ultimately yielding the triangle mesh required for rendering.

3.3. Limitations of the standard pipeline

Due to the lack of adequate consideration for the specific characteristics of PCB geometric elements and the employment of serial modeling algorithms, the modeling efficiency of the standard pipeline is notably low. Furthermore, the standard pipeline encounters considerable performance bottlenecks due to the large volume of triangle mesh data, resulting in high storage demands and insufficient rendering frame rates. The slow transfer of this data from the CPU to the GPU further exacerbates these inefficiencies, significantly hindering overall system performance.

4. Hybrid pipeline

Given the background, we proposed the hybrid pipeline. In our hybrid pipeline, the GPU is utilized for the majority of geometric computations, while the CPU is responsible for handling a few cases that are unsuitable for parallel processing due to their complexity.

4.1. Overview

In the hybrid pipeline(Fig.5), we categorize the geometries in the 2D model into procedural geometries (Trail, Circle) and non-procedural geometries (Area Shape). For procedural geometries, we design GPU-friendly data structures: LineLet, ArcLet, and TrailLet for Trail, and CircleLet for Circle. We also develop a viewpoint-driven dynamic modeling algorithm that adaptively constructs multi-level representations of these Let-structures on the GPU Sec.4.2.

For non-procedural geometries (Area Shape), we retain the extrusion algorithm from the standard pipeline to generate the Area Shell. We then introduce a GPU-based Delaunay convex hull triangulation algorithm [22] and adapt it to the triangulation of Area Shell in PCBs using a parallel triangle erasing algorithm4.3. Compared to traditional CPU-based triangulation methods, the GPU-based approach achieves a significant performance improvement.

This pipeline, which integrates different modeling approaches for procedural and non-procedural geometries, is also referred to as a hybrid pipeline .

4.2. Parallel Procedural Modeling

4.2.1 Data Structure

To better adapt to the parallel modeling algorithms in the mesh shader, we design four data structures: LineLet, ArcLet, CircleLet and TrailLet based on geometric characteristics (Fig.6). These data structures enable the modeling and triangulation processes for their corresponding geometric

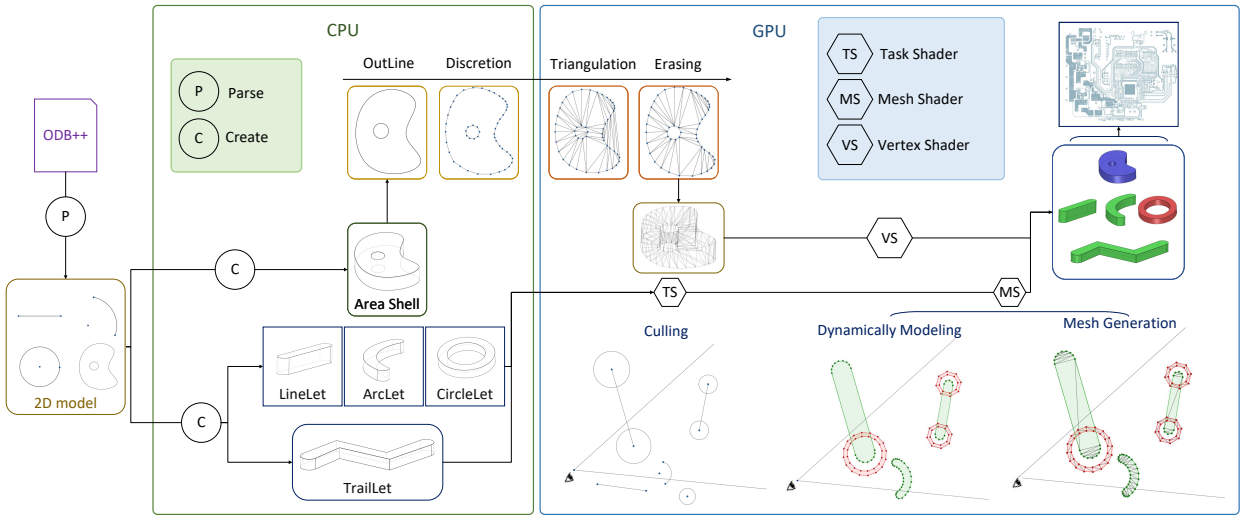


Figure 5. The hybrid pipeline modeling process categorizes 2D geometric data structures and processes four types of geometries: Circle, Line, Arc, Trail and Area. We encapsulate these four types of geometric bodies into a let-type data structure and use mesh shader for procedural parallel modeling. And mesh shader integrates dynamic modeling (LOD) and culling algorithms. For Area-type geometries, a GPU-accelerated Delaunay triangulation algorithm is employed to quickly generate triangle meshes, which are then rendered using the vertex shader. The process from a 2D model to rendering a single frame is referred to as PCB Modeling.

elements to be migrated into the mesh shader. The TrailLet is a geometric entity consisting of multiple line segments, whereas the other three types of Lets contain only a single geometric element.

The Lets-type data structure not only facilitates parallel computation in the Mesh Shader but also significantly reduces GPU input bandwidth compared to triangle mesh.

LineLet	**ArcLet**
HeadPoint	CenterPoint
TailPoint	HeadPoint
BaseHeight & TopHeight	TailPoint
Angle	BaseHeight & TopHeight
Width	Radius
TrialLet	Clockwise
PointsCount	**CirclelLet**
Points[]	CenterPoint
BaseHeight & TopHeight	BaseHeight & TopHeight
Width	Radius: outer & inner

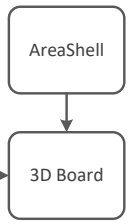


Figure 6. The data structure for several kinds of procedure PCB elements.

4.2.2 Modeling Methods

Based on the point coordinates in the Lets, we apply a perspective matrix transformation according to the viewpoint position to determine the Let's size in screen space .Based on different sizes, we generate and model Circles, Lines, and Arcs at different levels, as illustrated in Fig.7. Specifically, we use warp-level parallelism to process each Let structure. Within a warp, 32 threads work in parallel to generate 4, 2 or 1 vertices for the Let, depending on the level of detail.

Since the topological structure of each type of Let is identical, they can share the same mesh index. We preload the mesh indices for the three levels on the CPU and then transfer them to the GPU memory. After the vertex generation is completed, we use the corresponding mesh index to retrieve the mesh for the respective level.

Of course, this process only models geometries within the frustum; geometries outside the frustum are culled.

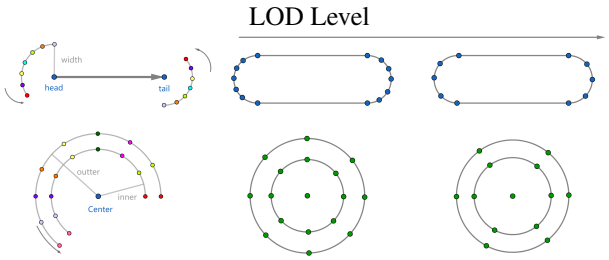


Figure 7. **LineLet** and **CircleLet** are modeled in parallel, with the same color using the same thread for vertex generation. Additionally, we generate a different number of points based on the LOD.

LineLet: LineLet is dynamically modeled into three levels, with the number of vertices at each level being 64, 32, and 16 respectively. The vertices of the top and bottom faces are each half of the total. The vertices generation process can be described by the following formula:

$$\alpha_i = \frac{i \cdot \pi}{level} + Angle \tag{4}$$

$$i = 0, 1, ..., 2 \cdot level + 1; \quad level \in \{15, 7, 3\}$$
$$P_i = Point + Width \cdot \begin{bmatrix} \cos(\alpha_i) \\ \sin(\alpha_i) \end{bmatrix} \tag{5}$$

Where i represents the thread index, α_i is the angle, and P_i is the 2D point generated by thread i . $Angle$ and $Width$ are the angle offset and width given in the let (Fig.6). By adding the height of the bottom or top face to the point coordinates, the complete vertices corresponding to the LineLet model can be obtained.

ArcLet & CircleLet: Similar to the handling of LineLet, the vertex generation algorithm is modified to read the arrays in ArcLet and CircleLet, and generate the corresponding three-dimensional discretized geometries.

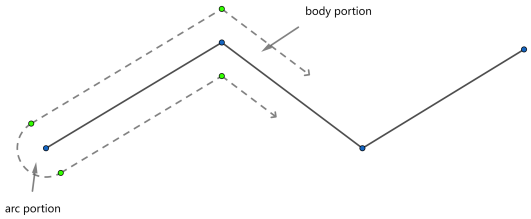


Figure 8. Bounding Process of the TrailLet

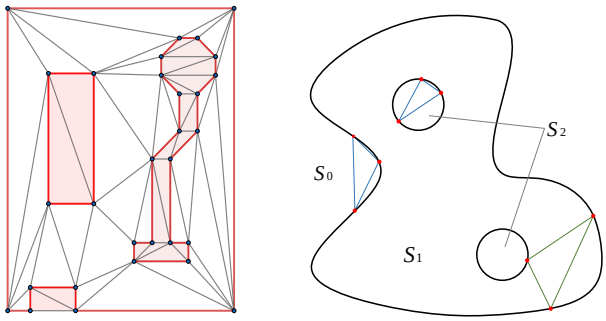
TrailLet: The generation of TrailLet is divided into two parts in Fig.8 : one for the arcs at both ends of the Trail and the other for the nodes in the middle of the Trail. The handling of the arcs at both ends is similar to that of LineLet, where each thread generates points along the arc at the endpoints. The nodes are distributed among the threads in the thread block based on the length of the Trail, and each thread processes one node until all nodes are handled.

Using the width and the two connecting line segments of the Trail, the new positions of the nodes are calculated after translation. The arc points and the nodes are then connected sequentially to form the 2D outline of the TrailShell.

The mesh of the TrailShell consists of two parts: The mesh for the arc portion is fixed; for any length of the Trail, an arc is always generated at the endpoints. The main body of the mesh, however, follows a regular pattern similar to a triangle strip. The MeshShader generates a corresponding length mesh based on the given Trail length.

4.3. None-Procedural Modeling

The modeling of non-procedural geometries is similar to the traditional pipeline. However, The existing GPU triangulation algorithms can only generate the convex hull triangulation mesh of the input point set and edge set, as shown in the Fig.9 (a) . Clearly, this does not fit our requirement for triangulating a polygon with holes. To accelerate the mesh retrieval process for Area Shell, we propose a parallel triangle erasing method that allows the GPU triangulation convex hull algorithm to be adapted to the PCB application scenario.



(a) (b)

Figure 9. (a) The red line segments represent the input boundary, and the pink area is the region to erase. (b) S_0, S_1, S_2 refer to the outer area, inner area and area in the hole respectively. According to the algorithm in Sec.4.3, triangles are checked in parallel and erased if they are in region S_0 or S_2.

Algorithm 1 Triangle Erasing

Initialize hash table H
for each vertex v_i **do**
  Insert vertex index and the loop value into hash table H
  $H[v_i] = loop_{value}$
end for
Parallel traverse each triangle T_j
for each triangle T_j **do**
  $v_1, v_2, v_3 \in T_j$
  if $H[v_1] == H[v_2] \& H[v_2] == H[v_3]$ **then**
    Check if the triangle is inside S_1
    $P = \sum_{k,i}^{0,1,2} \lambda_j P_i \quad \left(\sum_{j=0,1,2} \lambda_j = 1\right)$
    if $P \in S_1$ **then**
      Retain triangle T_j
    else
      Remove triangle T_j
    end if
  else
    Retain triangle T_j
  end if
end for

Triangulation and Erasing on GPU: As shown in Fig.9 (b), The existing algorithm converts the region S into a set of triangles T_i . Therefore, S_k can be expressed as follows: $S_c = \sum S_k, (k = 0, 1, 2) = \bigcup_i^n T_i$. So we need eliminate triangles outside of region S_1. Given the constraints on input edges and the fact that triangles do not overlap: $\bigcap_i^n T_i = \emptyset$ We can derive the following two formulas:

$$T_i^k \in S_k, \quad (k = 0, 1, 2) \tag{6}$$

Table 2. A performance comparison between the Allegro17.3 Standard Pipeline and our Hybrid Pipeline.

Model	PCB Size (kpin)	Triangle Number	Modeling Time (s)		Rendering time (ms)		Storage (MB)	
			Allegro17.3	**Our**	Allegro17.3	**Our**	Allegro17.3	**Our**
t1	0.88	1,130,956	5.9	**0.09**	0.41	**0.33**	116.2	**16.1**
t2	1.41	2,492,780	20.1	**0.28**	0.53	**0.46**	129.8	**42.6**
t3	2.54	10,579,900	22.9	**0.38**	1.41	**0.83**	998.2	**110.5**
t4	5.77	24,145,056	78.4	**0.68**	2.63	**2.01**	2973.7	**100.4**
t5	3.91	27,670,832	38.5	**0.75**	3.12	**2.22**	1946.8	**215.5**
t6	12.8	51,084,924	80.6	**1.35**	6.25	**3.85**	6023.6	**331.2**
t7	7.4	79,204,340	91.2	**1.79**	8.69	**4.76**	8310.1	**809.1**
t8	50	0.2 billion	322.5	**6.26**	16.67	**7.69**	19815.0	**1987.8**
t9	100	0.4 billion	–	**11.83**	–	**15.38**	–	**4021.3**

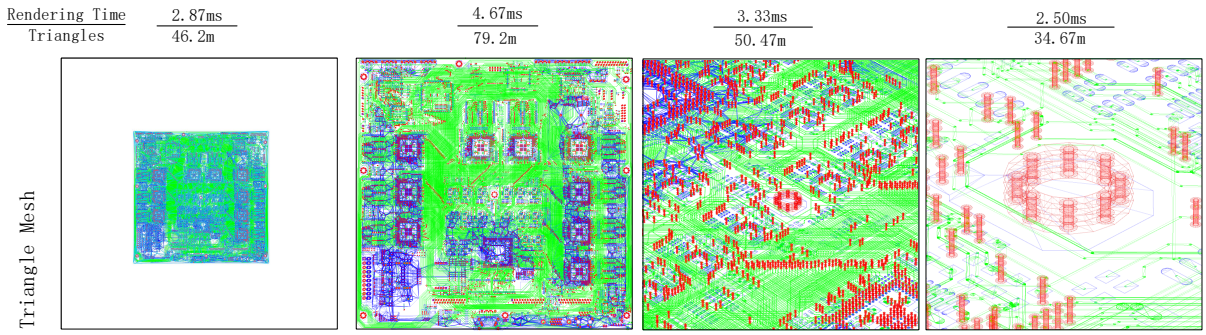


Figure 10. Triangle Meshes with different levels generated by our Hybrid Pipeline.

$$S_c = \bigcup_i^n T_i = \bigcup_i^n T_i^k, \quad (k = 0, 1, 2) \tag{7}$$

From these, we obtain: $\sum_i^{n_k} T_i^k = S_k$, n is the number of triangles . The necessary condition for a triangle to belong to S_k is : $P_\tau \in S_k \quad (\tau, k = 0, 1, 2)$ where P_τ represents the three vertices of the triangle. We use a hash table H to store the ownership of point P_i, and treat this necessary condition as a precondition for the subsequent judgment. If this condition is met, we can check whether the triangle T_i belongs to S_k by verifying if the centroid P of the triangle lies within S_k. The centroid P is calculated as:

$$P = \sum_{k,i}^{0,1,2} \lambda_j P_i \quad \left(\sum_{j=0,1,2} \lambda_j = 1 \right) \tag{8}$$

Finally, we get:

$$\bigcup_i^{n_k} T_i^k = S_k \tag{9}$$

By checking the case for $k = 1$, we can derive the required triangle mesh T^1. The algorithm is shown in Algorithm 1, loop refers to the outline or hole.

5. Result and Discussion

5.1. DataSet and Environment

We collected 300 PCB 2D models of varying sizes. In PCB engineering, the complexity of a PCB is typically described by the number of pins in a PCB model. It is positively correlated with the number of triangles after the model is discretized. Of course, the number of layers and coppers on the PCB also influence the number of triangles.

We implemented all algorithms using Vulkan 1.3 and Cuda12.6, and all test experiments were conducted on an i9-9900K CPU and an RTX 2080 Ti GPU. We selected nine circuit boards (t1-t9) of gradually increasing sizes as examples to demonstrate various scales of the algorithm.

5.2. Experimental Results

Tab.2 presents the key metrics of each algorithm, including rendering time and modeling time performance. We demonstrate the superior time performance of our algorithm by comparing it with Allegro 17.3, a commonly used EDA tool in the industry.

Model t9 (Fig.11) is the largest and most geometrically complex PCB in our dataset of conventional circuit boards. As shown in the Tab.2, Our hybrid pipeline completes the

modeling task within 11.83 seconds, while Allegro 17.3 fails to successfully model due to GPU memory limitations. Similarly, our rendering system also demonstrates excellent performance in terms of rendering time when rendering large-scale circuit boards.

5.2.1 Procedural and Non-procedural Geometries

In PCB engineering, more than 85% of the geometries are procedural, as shown in the Tab.3. Compared to the CPU serial modeling in the standard pipeline, we utilize GPU parallel modeling for the majority of the geometries, which undoubtedly significantly improves the modeling efficiency (Fig.11).

Table 3. Statistical data for different types of geometric elements. In PCB design, arcs are rarely used for routing.

Model	Line	Arc	Circle	Trail	AreaShell
t1	539	0	1395	529	823
t2	347	132	2457	1126	2964
t3	1455	0	14583	1334	2814
t4	1057	0	22505	3140	3740
t5	2401	86	33122	2693	5919
t6	7107	332	44168	6542	12601
t7	10801	580	92936	3105	6438
t8	28900	0	291660	26680	56280
t9	57800	0	583320	53360	112560

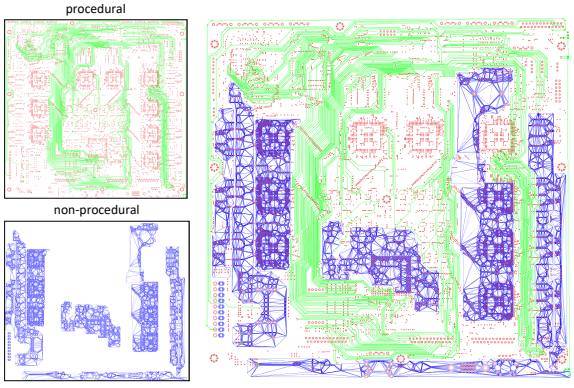


Figure 11. A PCB is composed of both procedural and non-procedural geometries, with procedural geometries accounting for more than 85% of the total.

5.2.2 Modeling and Mesh Generation

The modeling process refers to the complete workflow from reading the PCB data file to rendering the fully modeled PCB on the screen. This process consists of four components: file parsing, Let creating, curve discretization and triangle mesh generation. After testing, it was found that despite parallel processing, Let creating and mesh generation still accounted for the majority of the time overhead, as shown in the Tab.4.

Table 4. The time statistics for each part of the modeling process, with the time unit being milliseconds (ms).

Model	Parse	Create	Disc.	Tri.
t1	4.1	15.1	4.6	48.4
t2	10.2	72.1	10.1	138.7
t3	23.6	60.5	22.7	224.9
t4	63.9	124.9	42.5	446.9
t5	49.7	71.7	18.8	294.4
t6	100.4	245.7	66.5	840.6
t7	139.5	342.4	133.2	1151.1

5.2.3 Dynamic Modeling and Rendering

As shown in Fig.10, the dynamic modeling and rendering culling algorithms have significantly reduced the number of triangles in the scene, improving both storage efficiency and rendering performance. By selectively discarding unnecessary or out-of-view geometry, these algorithms minimize the computational load, allowing for faster processing and more efficient memory usage. This reduction in triangle count plays a key role in optimizing rendering speed and system responsiveness, especially for large and complex models. The following Tab.5 illustrates the impact of the culling and LOD algorithms on time performance.

Table 5. The table compares the rendering time performance with and without Frustum Culling and LOD optimizations (Opt. vs. W/O Opt.).

Model	Allegro17.3 (ms)	Hybrid Pipe (ms)	
		W/O Opt.	Opt.
t1	0.41	0.37	0.33
t2	0.53	0.47	0.46
t3	1.41	1.12	0.83
t4	2.63	2.50	2.01
t5	3.12	2.86	2.22
t6	6.25	4.76	3.85
t7	8.69	7.14	4.76
t8	16.67	15.3	7.69
t9	-	33.3	15.38

5.2.4 CPU/GPU Triangulation and Erasing

Compared to CPU-based triangulation, GPU parallel triangulation with our triangle erasing algorithm further reduces the overall modeling time. As shown in the Tab.6, the triangulation time cost reduced by 80% compared to the CPU-based triangulation. Due to the limitations in data transfer speed between the CPU and GPU, the performance

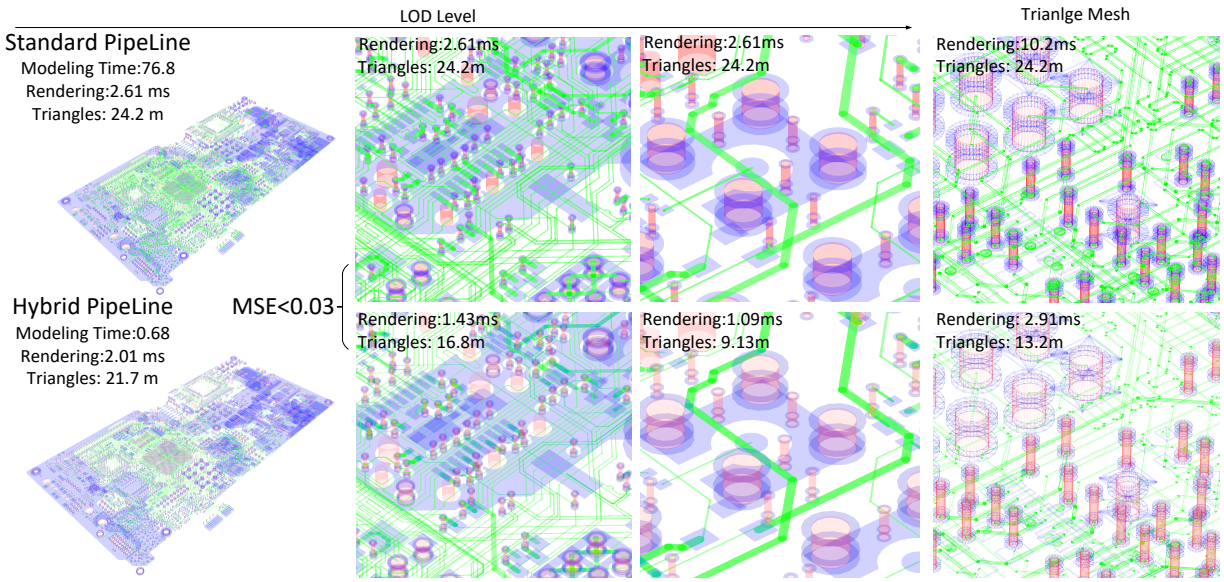


Figure 12. Comparison Between the Hybrid Pipeline and the Standard Pipeline in Terms of Modeling Speed, Triangle Number, Rendering Time and Rendering Quality (MSE).

Table 6. Comparison of Triangulation and Erasing Computation Time Between CPU and GPU.

Model	CPU Tri. (ms)	GPU Tri. (ms)
t1	67.2	48.4
t2	76.3	128.7
t3	1032.8	224.8
t4	3624.1	295.4
t5	5128.7	442.6
t6	10180.0	840.2
t7	10521.6	1121.5
t8	19430.8	5123.1

gap between the GPU parallel triangulation algorithm and the CPU-based serial algorithm is not significant for small-scale datasets. However, as the data size increases, the computational advantage of the GPU approach becomes increasingly evident, leading to a substantial widening of the time performance difference.

5.2.5 Accuracy and Correctness

In the field of PCB engineering, accuracy and correctness are two critical indicators. The generated 3D geometry must correspond precisely to the 2D geometry in the design and the geometric properties of the 3D model must meet the precision required for simulation and production. To demonstrate that the 3D models generated by our algorithm meet the industrial production accuracy requirements, we designed an experiment.

Since our design is based on the BRep data structure, it is straightforward to export the 3D data structure into a Step file, a format commonly used in CAD software (Sec.2.1). For the same PCB, we modeled it using both Allegro 17.3 and our algorithm, generating two different Step files. By reading these Step files with OpenCascade, we obtained two distinct models. We compare these two models through random sampling. Based on our experiments, the similarity between the Allegro 17.3 models and our models was above 99.1% for all the files tested, leading us to conclude that our algorithm is accurate and suitable for industrial PCB production.

Furthermore, after incorporating dynamic rendering and culling algorithms, the rendering accuracy of our hybrid pipeline meets the required standards, ensuring both efficiency and precision in the final output (Fig.12). Our experiments demonstrate that, across multiple test cases, the grayscale mean squared error (MSE) between two PCB renderings from the same viewpoint remains consistently below 0.03, as shown in the figure, indicating a high level of visual consistency between the results.

5.3. Limitations

Our algorithm performs excellently in terms of performance meets the accuracy and correctness requirements for industrial production, but there are still some shortcomings. First, due to the limitations of single-device GPU memory, our method may fail to process the 200k pin scale PCB till now, which is known the largest PCB. To address this issue, we can incorporate distributed solution with multiple GPU nodes. Secondly, our current discretization algorithm for NURBS curves is relatively simple. We can introduce nu-

merical matrix methods and other optimization techniques to accelerate NURBS curve computation and reduce redundant calculations during the recursion process [31]. These improvements will enhance the algorithm's performance in a broader range of applications.

6. Conclusion

In this paper, we proposes a hybrid pipeline, aimed at improving the efficiency, accuracy and rendering performance in the PCB modeling process. By leveraging GPU acceleration, we have achieved efficient modeling of large-scale PCB boards and made significant optimizations in the rendering stage. By employing culling and Level of Detail (LOD) algorithms, we have significantly increased the rendering frame rate, demonstrating exceptional performance, especially when handling large-scale PCB boards. Experimental results show that the hybrid pipeline offers much higher time efficiency and more excellent frame rates for modeling and rendering complex geometries compared to standard pipeline. Furthermore, a comparison with the industry-standard tool Allegro 17.3 further validates the superior time performance of our solution.

Nevertheless, there is still room for improvement in this method, particularly in introducing distributed rendering to support larger-scale scenes and enhance the system's overall scalability and performance. To avoid the computational overhead introduced by triangulation-based algorithms, it would be beneficial to include further discussions on real-time rendering techniques for algebraic surfaces without triangulation[19]. Future work will focus on further optimizing the algorithm, improving rendering quality, and exploring mesh-free or hybrid rendering strategies, as well as extending the proposed approach to broader application scenarios, with the goal of providing more efficient and precise solutions for PCB design and 3D modeling.

References

[1] Cadence Design Systems. Allegro PCB Designer, 2020. Software accessed: October 1, 2024. 1

[2] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan. A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 47–54, 2014. 3

[3] M. Â. A. d. Carvalho. *Exploring mesh shaders*. PhD thesis, 2022. 2

[4] Z. Chen, M. Qi, and T.-S. Tan. Computing delaunay refinement using the gpu. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, New York, NY, USA, 2017. Association for Computing Machinery. 3

[5] S.-W. Cheng, T. K. Dey, J. Shewchuk, and S. Sahni. *Delaunay mesh generation*. CRC Press Boca Raton, 2013. 3

[6] L. P. Chew. Constrained delaunay triangulations. In *Proceedings of the Third Annual Symposium on Computational Geometry*, SCG '87, page 215–222, New York, NY, USA, 1987. Association for Computing Machinery. 3

[7] S. EDA. Getting started with odb++ design odb++ viewer edition. 2023. 2

[8] H. R. Elliott. *Using Mesh Shaders for isosurface extraction*. PhD thesis, The University of Waikato, 2022. 2

[9] Y. S. Elshakhs, K. M. Deliparaschos, T. Charalambous, G. Oliva, and A. Zolotas. A comprehensive survey on delaunay triangulation: Applications, algorithms, and implementations over cpus, gpus, and fpgas. *IEEE Access*, 12:12562–12585, 2024. 3

[10] S. Fortune. Voronoi diagrams and delaunay triangulations. In *Handbook of discrete and computational geometry*, pages 705–721. Chapman and Hall/CRC, 2017. 3

[11] R. F. Francesco Buonamici, Monica Carfagni. Reverse engineering modeling methods and tools: a survey. *Computer-Aided Design and Applications*, 15(3):443–464, 2018. 1

[12] M. Khairy, A. G. Wassal, and M. Zahran. A survey of architectural approaches for improving gpgpu performance, programmability and heterogeneity. *J. Parallel Distrib. Comput.*, 127(C):65–88, May 2019. 1

[13] A. Köhn, J. Klein, F. Weiler, and H.-O. Peitgen. A gpu-based fiber tracking framework using geometry shaders. In *Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling*, volume 7261, pages 508–517. SPIE, 2009. 2

[14] M. Kraemer. Using turing mesh shaders: Nvidia asteroids demo, 2018. Accessed: 2024-12-09. 2

[15] C. Kubisch. New rendering techniques for real-time graphics: Turing - mesh shaders. In *SIGGRAPH*, 2018. Talk. 2

[16] C. Kubisch. Using mesh shaders for professional graphics, 2020. Accessed: 2024-12-09. 2

[17] D.-T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980. 3

[18] X. Li, G. Liu, S. Sun, and C. Bai. Contour detection and salient feature line regularization for printed circuit board in point clouds based on geometric primitives. *Measurement*, 185:109978, 2021. 1

[19] C. Loop and J. Blinn. Real-time gpu rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, July 2006. 10

[20] R. Mukundan. *The Geometry Shader*, pages 73–89. Springer International Publishing, Cham, 2022. 2

[21] S. Patidar, S. Bhattacharjee, J. M. Singh, and P. Narayanan. Exploiting the shader model 4.0 architecture. *Center for Visual Information Technology, IIIT Hyderabad*, 2006. 2

[22] M. Qi, T.-T. Cao, and T.-S. Tan. Computing 2d constrained delaunay triangulation using the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, page 39–46, New York, NY, USA, 2012. Association for Computing Machinery. 3, 4

[23] A. G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys (CSUR)*, 12(4):437–464, 1980. 2

[24] J. N. Rodriguez, M. C. Canosa, and E. H. Pereira. Improving electrical power grid visualization using geometry shaders. In *2011 Eighth International Conference Computer Graphics, Imaging and Visualization*, pages 177–182, 2011. 2

[25] B. Santerre, M. Abe, and T. Watanabe. Improving gpu real-time wide terrain tessellation using the new mesh shader pipeline. In *2020 Nicograph International (NicoInt)*, pages 86–89, 2020. 2

[26] M. A. M. Sathiaseelan and N. Asadizanjani. Highly accurate and portable 3d surface analysis tool (apsa) for printed circuit boards (pcb) reconstruction and assurance. *Microscopy and Microanalysis*, 27(S1):784–787, 2021. 1

[27] J. Schertzer, C. Mercier, S. Rousseau, and T. Boubekeur. Fiblets for real-time rendering of massive brain tractograms. *Computer Graphics Forum*, 41:447–460, 05 2022. 3

[28] J. J. Shah and M. Mantyla. *Parametric and Feature Based CAD/Cam: Concepts, Techniques, and Applications*. John Wiley & Sons, Inc., USA, 1st edition, 1995. 2

[29] J. R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry*, 22(1-3):21–74, 2002. 3

[30] T. Tricard. Interval shading: using mesh shaders to generate shading intervals for volume rendering. *Proc. ACM Comput. Graph. Interact. Tech.*, 7(3), Aug. 2024. 3

[31] R. Xiong, Y. Lu, C. Chen, J. Zhu, Y. Zeng, and L. Liu. Eter: Elastic tessellation for real-time pixel-accurate rendering of large-scale nurbs models. *ACM Trans. Graph.*, 42(4), July 2023. 2, 10