Dynamic Unwrapping and Meta-Shaders

University of the Fraser Valley
Abbotsford, British Columbia, Canada
unbounded.research.team@gmail.com

Eli Landa
Vancouver, British Columbia, Canada
unbounded.research.team@gmail.com

ABSTRACT

We present an implementation of meta-shaders by adapting a commonly known data structure to pass into shaders: planar graphs. This allows for an arbitrary flexibility in design where the face regions that partition a surface can be individually and uniquely shaded, or in combinations, as desired. Considering a wide variety of systems mixing 2D and 3D across multiple industries, the applications of meta-shaders opens up a new area of design and user-interaction possibilities in computer graphics.

CCS CONCEPTS

• **Computing methodologies** → **Texturing**; *Procedural animation.*

KEYWORDS

planar graphs, scalar-vector graphics, texture unwrapping, shaders

ACM Reference Format:
and Eli Landa. 2023. Dynamic Unwrapping and Meta-Shaders. *ACM Trans. Graph.* 1, 1 (April 2023), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

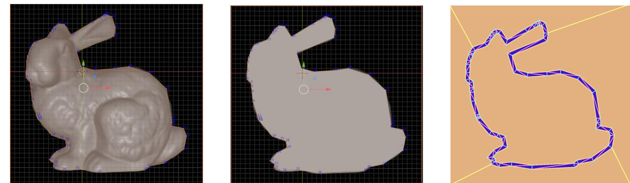

Figure 1: (left) The Stanford Rabbit. (middle) Flat geometry overtop rabbit's profile. (right) Shader demonstrating both a planar graph structure partly built from the flat geometry, and binary search to colour its face regions (without the typical quadtree or other distance query structures); the graph has cyan vertices, orange edges, and blue or red face colours.

1 INTRODUCTION

In the world of computer graphics there is a problem of rigidity when it comes to colouring the surface of 3D geometries. Most approaches combine formulae and involve dividing space in some manner to create a pattern. This can limit the possibilities that an object can be shaded both in artistic terms and as well as in precision. Another common way in which geometry is shaded is via raster textures. While rasterized textures allow for seemingly arbitrary colouring, this is not the case. Raster images are composed of pixels and if the camera moves too close to the geometry's surface—discontinuity between individual pixels will result. The solution we propose in this paper will mitigate these issues of rigidity, and discontinuity by presenting a new way to colour geometry using a technique we call meta-shaders.

In computer science and mathematics, a well-known concept of a graph is composed of vertices, edges, and faces. A graph has no preset positions in 3-dimensional space, unlike a 3D mesh. However, by passing the data of a graph to a shader, we assign 2D positions to the vertices of the graph on the surface of a mesh. Thus, we create colourings without discontinuities, and with endless precision. Effectively this approximates scalar-vector graphics (SVG), with the additional ability to perform binary search across a surface. Furthermore, since our shader is relying on a graph structure, it is effectively displaying meta-geometry (as opposed to mesh geometry). We can then apply a shader to the different face regions for rendering. This enables artists to produce effects such as animation, dynamically colour the face regions of the graph, and increase the opportunities for functionality with interactivity and design.

With meta-shaders, we can update the regions to display in other shaders conditionally. As the environment changes and interacts with our object, we may want to move the region for real-time effects or change the shader behaviour for a specific face, or group of faces. With our approach, all of this and more is possible.

1.1 Requisite Knowledge

A graph G is a set $V(G)$ of vertices with a set $E(G)$ of pairs of vertices called edges such that $E(G) \subseteq \big\{\{u,v\} \,|\, u,v \in V \text{ and } u \neq v\big\}$. Vertices are called adjacent if an edge connects them. A graph is considered planar if there exists at least one placement of vertices (called a drawing) in the plane such that no edges are crossing. A drawing of a graph on some surface with no crossing edges is called an embedding. As such, a drawing of a graph on the plane with no crossing edges is called a planar embedding. The regions bounded by edges in a planar embedding are called faces, but we use the term region to avoid confusion with mesh geometry.

As an important note, if a graph is planar—that is, the graph can be drawn on the plane with no crossing edges—then that graph can be drawn on any surface with no crossing edges.

1.2 Embeddings of Planar Graphs

A cyclic ordering of the adjacencies in each vertex's adjacency list for a graph defines what is called a combinatorial embedding. We are able to check which surface a graph embeds by visiting the

Authors' addresses:, University of the Fraser Valley, Abbotsford, British Columbia, Canada, unbounded.research.team@gmail.com; Eli Landa, Vancouver, British Columbia, Canada, unbounded.research.team@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
0730-0301/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Draft

vertices in the order of a face-walk. A face-walk consists of visiting vertices in sequence based on the cyclic ordering of adjacencies [Boyer and Myrvold 1999] managed by a record data structure for each side of an edge used to walk. Using face-walking with records is a simpler approach to implement checking whether the embedding of a graph is planar. Then further methods can be used to obtain a drawing.

Tutte embedding is one of the first well-known efficient methods to draw a planar graph with straight-line edges between vertices [Tutte 1963], which uses Barycentric coordinates and requires a face of the graph to have preset positions which form the boundary of a convex polygon.

There exist a large number of algorithms for embedding a planar graph, but many are complex and have trade-offs between difficulty in implementation and the utility of the result. Currently, our prototype system relies on the user to design the connections to place the vertices in locations that make it an embedding without edges crossing. Some ways to expedite the design is to import an object file from Blender and edit further in Unity 3D. We will explore implementations of algorithmic embedding in the future that help automate the design of a planar graph further.

1.3 Related Work

Groups and Graphs [Kocay 2007] was software developed to embed graphs on surfaces such as the torus and Klein bottle with visualizations that aided reading cyclic adjacency orders. This mainly supported mathematicians with research of combinatorial embeddings on such surfaces.

A fundamental way to make location queries on a 2D surface is with quadtrees [Samet 1984]. Quadtrees can be implemented efficiently using binary interleaving [Redis 2022], but only to a resolution of cells—or, square regions—where the area of queries have some radial limit based on the combined area together with neighbouring regions.

Visualization of graphs do not necessarily have to be planar, and open source web development libraries—such as D^3 [Bostock et al. 2011]—provide tools for fast interactive 2D display of large graphs in browsers.

Applications and systems mixing 2D and 3D benefit from potentially more intuitive ways of user-interaction, such as converting sketch-like input to quickly mock-up architectural designs [Olivier et al. 2019]. The open source community benefits greatly from Blender's suite of tools, with Grease Pencil having a thoroughly developed and robust set of features that generates flat geometry to display strokes, among other effects [Leung and Lara 2015].

But we have found little in the way of development of shaders that display SVG, although we have seen plenty of demonstrations of geometric shapes displayed using surface-distance functions, which are known to be costly for the iterations involved with stepping through space to form their shapes.

2 DYNAMIC UNWRAPPING

2.1 The Goal

We construct a system such that regions of a planar embedding can be mapped to nearly any polygonal shape regardless of complexity to effectively partition a surface. Further, the regions of the embedding can be rendered separately with different shaders using only logarithmic time binary search per fragment to find which region a pixel resides.

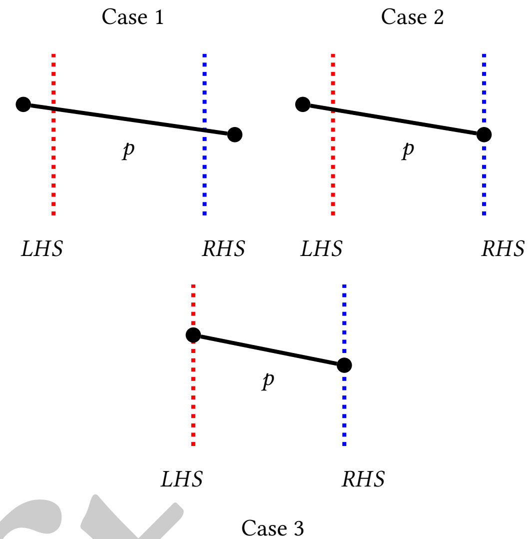

Figure 2: The three possible cases for edges intersecting strips.

2.2 Process

The basis of our approach depends on creating vertical "strips" of the plane surface for each vertex in the graph. The data for these strips is passed via texture to the shader. The texture can then have binary search lookups depending on any point on the plane to find which region that point belongs to. The advantage of using vertical strips is that there is a clear order to each of the vertices by the x-coordinate. Vertices with the same x-coordinate belong to the same strip. This will let us sort the vertices accordingly in which we can logarithmically find the two vertical lines L_1 and L_2 between which the point lies. Thus the face for which this point is on must cross some non-empty subset of this space bounded by L_1 and L_2.

The core assumption that will let us determine a region of the planar embedding is as follows. For closest pairs of vertices, say v_1 and v_2 positioned at points (x_1, y_1) and (x_2, y_2) on the plane, respectively, it must be that for $\{v_1, v_2\} \in E(G)$ and $x_1 < x_2$, no other vertex to the right of x_1 has x-coordinate less than x_2. Thus, it cannot be the case that a vertex of an edge passing through strip between the vertical lines $x = x_1$ (LHS) and $x = x_2$ (RHS) can contain another vertex inside this region. Thus, an edge has one of three possible ways to intersect a vertical strip:

(1) Passes through both lines:
  - neither endpoint vertices lie on either vertical line.
(2) Passes through one of the vertical lines only:
  - a vertex lies on one of the vertical lines.
(3) Passes through neither lines:
  - an endpoint on LHS and the other on RHS.

Draft

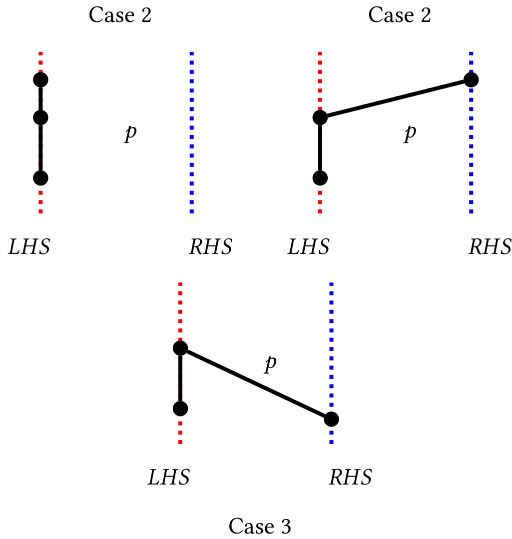

Figure 3: The three cases for an edge's interaction with the strip. Cases 2 and 3 partition the strip by either having both vertices completely above the pixel (Case 2) or by having one vertex above and the other below. These cases are equivalent to the cases when mirrored vertically so that the LHS vertex is below the pixel. Case 1 does not partition the strip.

We add four vertices to the graph, one vertex to each corner of the plane surface so that vertical strips cover the entire surface. Altogether, the collection of vertical strips for the entire surface allow us to binary search which strip a fragment sits.

Further, once we know the specific strip, we can binary search within the strip vertically by distance above each edge in the strip, since the edges of the graph in a strip partition the strip, except for the perfectly vertical edges along a strip's LHS. Once the vertical section is located for a fragment, we must perform a check to determine which of the three closest edges in the strip at that section should be used to determine the correct region of the embedding.

3 IMPLEMENTATION

3.1 Vertical Strip Construction

In Algorithm 1 we construct the vertical strips needed for binary search in our meta-shader.

3.2 CPU Runtime

The outermost loop iterates over each strip to collect the edges that intersect the strip.

There is a possibility for there to be, at worst—*all* edges intersecting *all* vertical strips—which results in a quadratic runtime. This could be avoided by building strips in some other partitioning scheme, such as building them to be horizontal instead, or to be radial from any desired point. Depending on the graph desired, one could choose a strip building strategy to keep the runtime close to linear w.r.t. the number of edges.

Note that to keep the runtime linear, we take advantage of fast copying of the immediately previous strip just built to the left provided by C# List, instead of copying element by element, of course.

ALGORITHM 1: We construct the vertical strips for binary search to query regions for the input plane embedding for graph G.

Input: A graph $G(V, E)$ with adjacency lists giving E such that the clockwise ordering of neighbours in the adjacency lists determine the embedding of G in the plane combinatorially.
Output: A collection of edge lists named strips, each list corresponding to one vertical strip, where the edges for each list are sorted in vertical ascending order.

1 Sort all vertices' x-values ascending, ignoring repeated values;
2 $S \leftarrow$ number of unique x-values;
3 Create an array of S lists strips;
4 Add all adjacencies of vertices with x-value 0 to strips[0];
5 (this completes the leftmost vertical strip);
6 **for** $i \leftarrow 0$ **to** $S - 1$ **do**
7   copy strips[i] to strips[i+*1*];
8   **for** *each adjacency* $e = \{u, v\}$ *in* strips[i] **do**
        `// v is righthand endpoint of adjacency`
9   **if** v *has* x*-value* <= x*-value of LHS for* strips[i+*1*] **then**
10   remove e from strips[i+*1*];
11   **end**
12   **end**
13     $k \leftarrow$ number of vertices that lie on the LHS of strips[i+*1*] with the same x-value;
14   **for** *each vertex* u *counted to get* k **do**
15   **for** *each edge* e *incident on* u *in the graph* **do**
16   **if** *e lay on top of or angled to the right of the LHS of* strips[i+*1*] **then**
17   Add e to strips[i+*1*] while maintaining clockwise ordering of adjacencies for u;
18   **end**
19   **end**
20   **end**
21   Replace vertical successive edges (no other edges in strips[i+*1*] between them) with *one* vertical edge;
22 **end**
23 **for** $i \leftarrow 0$ *to* S **do**
24   Sort the adjacencies of strips[i] ascending in y-value where each edge crosses the LHS of strip for strips[i+*1*];
25 **end**
26 Pass strips to shader;

Table 1: Early Prototype testing for our real-time meta-shader graphs.

	Count
Vertices	117
Edges	200
Vertical Strips	116
FPS	100

4 PERFORMANCE EVALUATION

Testing for our meta-shaders.

Draft

5 FUTURE WORK

5.1 Adaptations and Optimizations

Graphs in graphics are already ubiquitous as mesh geometry. RXMesh splits the geometry of meshes using algorithms of graphs as a data structure to aid optimizations for work on GPUs[Mahmoud et al. 2021].

[Tran and Cambria 2018] surveyed graph algorithms that take advantage of parallelization of GPUs. Contemporary work by [Jaiganesh and Burtscher 2018] improves efficiency to find connected components of graphs using GPU.

A survey of the prior work optimizing the rendering and use of meshes could be gleaned for possible performance increases in our system.

With procedural generation of the vertices and edges of a graph, the geometry of a mesh could be tessellated in real-time with the user making design decisions. Generating this way would avoid storing meshes with a large amount of data and save storage memory.

This work is in its early stages. As such there is much do be done to refine and optimize this process. Other restrictions or assumptions may lead to further improvements on our design and produce more optimal algorithms. Naturally, young research like this will produce new problems to overcome and many other research opportunities in the field.

5.2 Possible Applications

There are a large number of possible applications to consider across many industries, and we continue to search for more in a broad survey through the literatures. Three main application directions up to the time of writing we see as being feasible:

- meta-shader for flexibility in combining shaders
  - interactive and infinite-zoom SVG approximation on the surface of objects
- vector graphics features and interactivity on surfaces for 3D design tools
- as a data structure for managing location queries within regions

5.3 Meta-Shaders

So many tools have blazed the forefront of what is possible with generating 3D content through interactive mixes of 2D and 3D. A survey done by [Bhattacharjee and Chaudhuri 2020] lays out much comparison between them. We see such a wide variety of possible applications of our work that there are plenty of opportunities for advancements.

Because the system we are developing approaches the design of textures in a different way from anything listed so far, there is also opportunity for unique applications that tend to be purely developed as research within the entertainment industry, such as those described in [Mlynarčíková 2020]: 2D mixed with heavily stylized 3D for Spider-man: Into the Spider-verse.

The most straightforward application of meta-shaders is to combine with other shaders, say powerful design methods demonstrated by [Thonat et al. 2021] for tesselation-free displacement mapping, that could be used along with any other shader in any arrangement of face regions. We specifically cite this displacement technique as meta-shaders can be used on very low-poly surfaces where the design is moved to the face regions partitioning the surface, as opposed to relying on complex mesh geometry.

Another use is to provide interactivity and animation of the graph's regions in real-time. We have not seen examples of similar functionality of graphs in shaders.

5.4 Scalar Vector Graphics

To a more extreme set of possible uses, consider the fashion technology industry. Many interactive design applications have been compared in [Rizkiah et al. 2020]. There is no example currently that we can find for use of scalar-vector graphics in any of these applications. We are sure expert designers of fabrics and garments would find use for similar tools as those found in vector-graphics software, but on the surface of 3D models. Planar graphs can approximate SVG, and be regenerated given the distance of a camera to the surface of a 3D model to effectively provide infinite-level detail in textures.

Finally, while we read the work of [Leake et al. 2021], we were reminded of Ada Lovelace, who foresaw computer applications involving visualization and was inspired by the punch-cards of automated looms. [Leake et al. 2021] pulled inspiration from quilting for research that analyzed properties of paper-pieceable-quilt tile patterns, which partially correspond to planar graphs. It is an example of niche areas of design that could potentially benefit from our techniques.

5.5 Data Structures

Many have used various data structures for managing location queries in a 2D or 3D scene, such as quadtrees and octrees, as well as others such as bounded volume hierarchies (BVHs), or improvements on these that use bitwise interleaving to avoid the need to take up memory with the same advantages of hashing into the data structure faster than traversing directly from the root to the desired child node.

There is much more flexibility to design any regions in 2D using an embedded graph in the plane. The advantage of planar graphs is that they have a rich plethora of algorithms crafted by mathematics and computer science researchers for many possible applications.

For example, if one only needs collision detection directly surrounding a large object, one large region set up with a polygon and adjacent regions surrounding it of any design could provide various adjustments to the typical data structures used for collision detection. Visiting the edges along the boundary of a region allows one to also visit the neighbouring regions.

REFERENCES

Sukanya Bhattacharjee and Parag Chaudhuri. 2020. A Survey on Sketch Based Content Creation: from the Desktop to Virtual and Augmented Reality. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 757–780. https://doi.org/10.1111/cgf.14024

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

John M Boyer and Wendy J Myrvold. 1999. Stop Minding Your p's and q's: A Simplified O (n) Planar Embedding Algorithm.. In *SODA*. 140–146.

Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 92–104.

Draft

William Kocay. 2007. Groups & graphs–software for graphs, digraphs, and their automorphism groups. *Match* 58, 2 (2007), 431–443. https://match.pmf.kg.ac.rs/electronic_versions/Match58/n2/match58n2_431-443.pdf

Mackenzie Leake, Gilbert Bernstein, Abe Davis, and Maneesh Agrawala. 2021. A Mathematical Foundation for Foundation Paper Pieceable Quilts. *ACM Trans. Graph.* 40, 4, Article 65 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459853

Joshua Leung and Daniel M Lara. 2015. Grease pencil: Integrating animated freehand drawings into 3D production environments. In *SIGGRAPH Asia 2015 Technical Briefs*. Number 16. 1–4. https://doi.org/10.1145/2820903.2820924

Ahmed H Mahmoud, Serban D Porumbescu, and John D Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Transactions on Graphics* 40, 4 (2021). https://escholarship.org/uc/item/8r5848vp

Ema Mlynarčíková. 2020. Integration of 2D Animation into a 3D Pipeline. (2020). http://hdl.handle.net/10563/46827

Pauline Olivier, Renaud Chabrier, Damien Rohmer, Eric De Thoisy, and Marie-Paule Cani. 2019. Nested Explorative Maps: A new 3D canvas for conceptual design in architecture. *Computers & Graphics* 82 (2019), 203–213. https://doi.org/10.1016/j.cag.2019.05.027

Redis. 2022. *Multi-dimensional indexes.* Retrieved January 21, 2022 from https://redis.io/topics/indexes#multi-dimensional-indexes

M Rizkiah, I Widiaty, and B Mulyanti. 2020. Fashion pattern maker of software application development. In *IOP Conference Series: Materials Science and Engineering*, Vol. 830. IOP Publishing. https://doi.org/10.1088/1757-899x/830/4/042096

Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)* 16, 2 (1984), 187–260. https://doi.org/10.1145/356924.356930

Theo Thonat, Francois Beaune, Xin Sun, Nathan Carr, and Tamy Boubekeur. 2021. Tessellation-free displacement mapping for ray tracing. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–16.

Ha-Nguyen Tran and Erik Cambria. 2018. A survey of graph processing on graphics processing units. *The Journal of Supercomputing* 74, 5 (2018), 2086–2115. https://doi.org/10.1007/s11227-017-2225-1

William Thomas Tutte. 1963. How to draw a graph. *Proceedings of the London Mathematical Society* 3, 1 (1963), 743–767. https://doi.org/10.1112/plms/s3-13.1.743

Eli Landa,,

Draft