Grafi v računalniški grafiki

Žiga Lesar, Matija Marolt
University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, Ljubljana, Slovenia
ziga.lesar@fri.uni-lj.si, matija.marolt@fri.uni-lj.si

Izvleček

Pričujoče delo vsebuje pregled temeljnih idej iz teorije grafov, ki se uporabljajo v računalniški grafiki. Navedene metode se med drugim uporabljajo za stiskanje poligonskih mrež, v animaciji, pri določanju vidnosti, za optimizacijo upodabljanja, prepoznavo oblik in navigacijo. Predstavljamo tako uspešne kot neuspešne primere uporabe iz najbolj citiranih del in sodobnih objav. Pregled področja razkriva trend uporabe grafov za izluščenje bistvenih informacij iz danega konteksta, kar se neposredno izraža v obliki raznovrstnih kompresijskih shem in učinkovitih poizvedovalnih podatkovnih struktur.

Ključne besede: grafi, računalniška grafika, navigacija, povezanost, upravljanje vidnosti, stiskanje podatkov

Graphs in computer graphics

Abstract

This paper provides an overview of the main ideas of graph theory used in computer graphics. Applications include mesh compression, animation, visibility determination, rendering optimization, shape recognition, and navigation. We present both success and failure cases from the most cited works and recent publications. Related work review reveals a trend in using graphs for extracting the essential information from a given context, which directly manifests itself in various compression schemes and efficient querying data structures.

Keywords: Graphs, computer graphics, navigation, connectivity, visibility management, data compression

1 INTRODUCTION

This paper aims to introduce the topic of graphs and networks to computer scientists working primarily in the field of computer graphics. It provides an overview of the main ideas, the most groundbreaking work, and applications where graphs are the most natural representation of a given problem. The paper is divided roughly into two parts, the first part focusing on interactive graphics and object representation, which are especially relevant to interactive and real-time graphics, and the second part focusing on volume representation, point clouds and light transport simulation, which are mostly used in scientific applications and non-interactive graphics.

Graphs are a suitable representation for any data, in which relationships between objects or concepts are of essential importance. Compared to tables, arrays or textures, they encode more information about a given subject, albeit with a less rigid structure. High-level structures in graphs, such as connected components, cliques and spanning trees, may reveal certain aspects of the data, which cannot be sufficiently expressed in other representational forms. Therefore, in many cases in computer graphics and animation, graphs may be the most natural, the most informative, and/or the most expressive representation. For example, graphs are ideal candidates for pathfinding in video games, motion synthesis from motion cap-

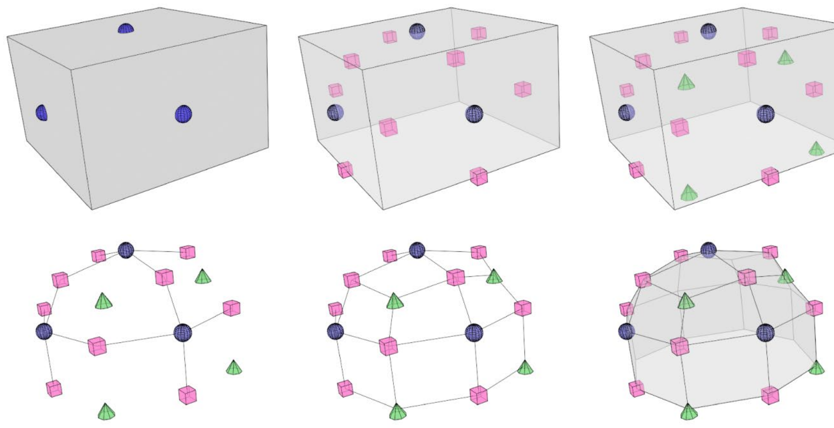

Figure 1: **The six steps of the Catmull-Clark subdivision scheme. A vertex is created for each face, vertex, and vertex from the original mesh, then the new vertices are appropriately connected together to form the subdivided mesh. Author: UserTwoSix, under CC BY-SA 4.0.**

ture in animation studios, and shape matching in 3D object databases.

Graphs come with their own drawbacks, however. They are more difficult to process (especially in parallel), they are inherently unordered, and high-level structures first have to be extracted or computed before use. In practice, these drawbacks are often mitigated by using graphs with a more rigid structure, such as octrees, or by using graphs to represent only a part of the data, and using other representational forms for the rest. Deciding what to represent with graphs and what not to, remains part of ongoing research and a practical art. This paper hopes to highlight some of the success and failure cases so that the reader can make an informed decision when tackling a new problem.

2 GRAPHS AND NETWORKS

In this work, we use the term graph to describe a discrete object composed of vertices and edges. Depending on the context, both vertices and edges may represent different things, from geometric primitives and their adjacencies to animation states and their composability. See the book Graphs on Surfaces by Mohar and Thomassen [18] for a comprehensive introduction on the subject. In the following sections, we list the most common applications in which graphs are used in the field of computer graphics. Each section first explains the problem at hand, then shows how graphs can be used to encode it, and finally, the most relevant ideas and publications that use such graphs for implementing the solutions to the problem.

2.1 Connectivity graphs

Modern graphics processing units (GPUs) are specialized hardware for displaying polygonal meshes, specifically triangular meshes. In real-time applications, such meshes are usually stored in on the GPU as an array of vertices together with an array of indices. The vertices are usually equipped with attributes describing, for example, their location in space, the normal of the surface at that location, or texture mapping information. The indices, on the other hand, describe the connectivity between the vertices, and may be used to represent points, edges, or triangles. The most direct mapping from polygonal meshes to graphs is to map mesh vertices to graph vertices, and edges of geometric primitives to graph edges. In 3D modeling applications, a winged edge representation [3] is much more common, since it simplifies many types of adjacency queries by including pointers to adjacent geometric primitives. In a winged edge representation, for example, all geometric primitives (points, lines, and faces) may be mapped to graph vertices, while graph edges may represent adjacencies.

Common tasks in the field of mesh representation include connectivity or geometry compression, optimizing the curvature of the surface, minimizing the deformation of the triangles, maximizing the internal angles of the triangles, and generating new vertices and triangles from a low resolution mesh.

One of the most popular algorithms in computer graphics is the Catmull-Clark subdivision algorithm [7], used to recursively generate smooth subdivision surfaces from a low-resolution mesh. Created by

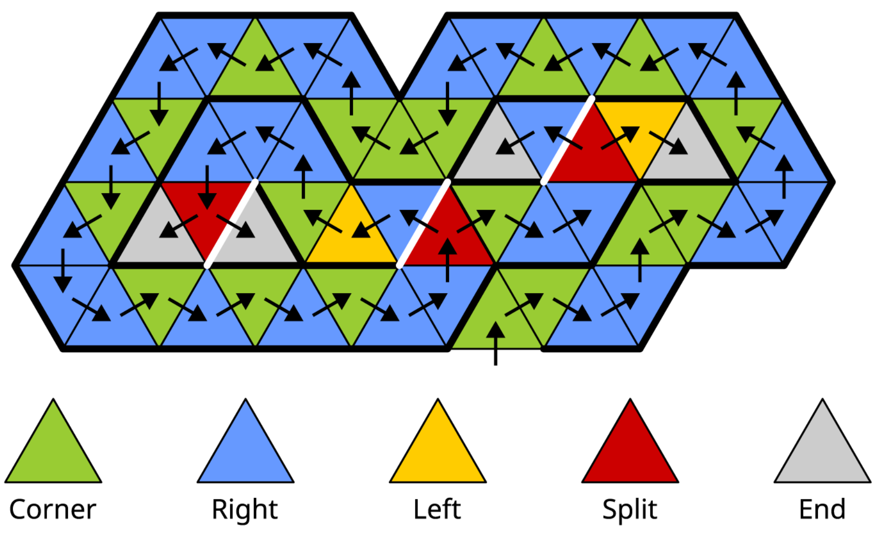

Figure 2: **Demonstration of the edgebreaker algorithm and the 5 different cases during traversal.**

scientists at Pixar, the algorithm is used heavily in the movie industry, where it significantly facilitates modeling of detailed meshes. The algorithm works by fitting B-spline surfaces to the control mesh, and then generating new points and geometric primitives from that B-spline. This approach relies heavily on the topology of the original mesh, since it is difficult to join the B-spline surface patches at vertices with odd degrees. Rectangular control-point meshes reduce to standard B-spline surfaces, which are shown to be continuous both in tangent and in curvature, whereas triangular meshes may include so-called extraordinary points, at which the surface can be shown to be continuous at least in tangent. A different algorithm, created by Doo and Sabin [10, 11], works similarly well on arbitrary meshes. A simpler algorithm was later created by Loop [16], with the intention of being efficient to evaluate. In fact, after programmable GPUs became widely accessible, the algorithm was shown to be efficient to evaluate on programmable GPU tessellation units [17]. Topologically, all the listed algorithms work in a similar fashion, by mapping a graph corresponding to a winged edge representation back to a mesh, effectively creating a mesh vertex for each geometric primitive in the original mesh (see Figure 1).

Meshes require notoriously large amounts of data to be accurately represented. Fortunately, surfaces usually exhibit a high degree of redundancy or predictability; therefore, compression can be used to efficiently transmit meshes, although an uncompressed form is required by the GPU. Usually, geometry information (e.g. positions, normals, tangents) is compressed separately from connectivity information (indices describing geometric primitives, such as triangles). The most widely used algorithm for connectivity compression is Rossignac's edgebreaker [22], used in the popular Draco mesh compression format. Edgebreaker permutes the list of vertices so that the connectivity mostly reduces to common patterns, such as triangle lists or triangle fans. The triangles can be further sorted so that geometric adjacency is mostly respected in the list. Given a triangle, any one of the three edges can be extended with respect to the parallelogram rule to generate a new triangle. Consequently, the list of triangles can be generated with a walk across adjacent triangles, where only one of five events can occur at each triangle (see Figure 2). The resulting list of events is highly compressible.

The accuracy of the above mesh reconstruction relies on the regularity of the mesh. The geometry of successive triangles, which closely match the predicted parallelogram, can be efficiently compressed. On the other hand, irregularities exhibit large deviations from the predictions and consequently require more data to be accurately reconstructed. This fact is exploited by Sorkine and Cohen-Or [25], who developed a method for mesh regularization for altering the geometry of a mesh so that it is more predictable and compressible. The method retains the connectivity information, but alters the geometry in the least squares sense by choosing a set of control points from the existing vertices and then solving a sparse linear system to compute the positions of the

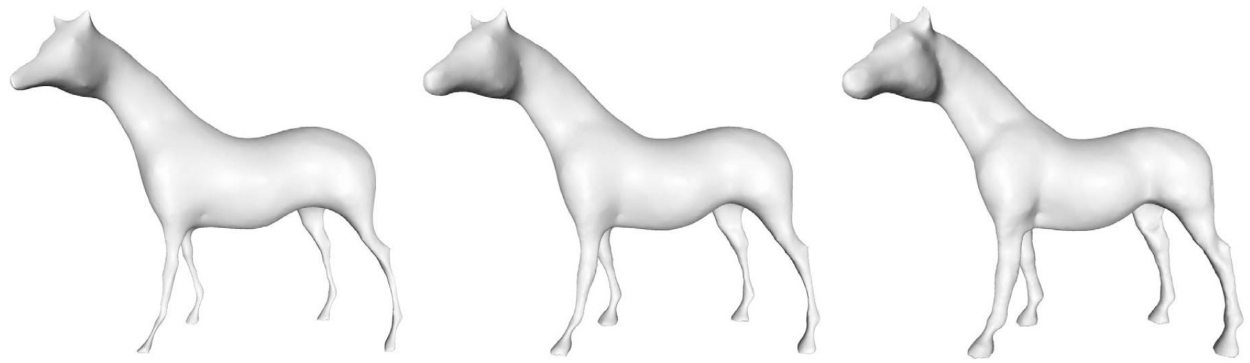

Figure 3: **Least-squares meshes with varying amounts of control points: 200 (left), 1000 (center), and 3000 (right). Image courtesy of O. Sorkine [25].**

rest of the vertices. The constraints can be weighted to balance the control point error and fairness (which corresponds to the regularity of the mesh). Naturally, more control points corresponds to a more detailed mesh (see Figure 3).

For further information on connectivity graphs and mesh compression, we refer the reader to the survey paper by Alliez et al. [2], where the authors discuss several single-rate and progressive schemes, along with the optimality of the approaches.

2.2 Skeletons

Connectivity graphs are often much too complex for high-level tasks, such as animation and shape matching, for which simplified representations are usually preferred. Such simplified, high-level representations preserve only the most fundamental information about the overall shape of the mesh and disregard any detailed geometry and connectivity information. Generation of such representations is closely related to the field of computational topology.

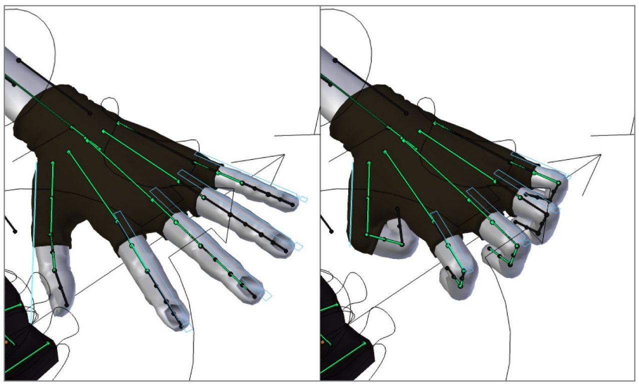

Figure 4: **An example skeleton of a hand. Source: Blender Foundation, under CC BY-SA 3.0.**

In interactive computer graphics, a very common technique for mesh animation is through skeleton manipula- tion. Skeletons represent the fundamental shape of a mesh with a set of bones connected with joints (see Figure 4). A bone is simply a rigid transformation relative to the parent bone, whereas a joint represents how the bones are connected together to form a structure. In graph terminology, a bone is a vertex and a joint is an edge, and a skeleton is a tree. When a bone moves, all descendants of that bone are affected by its transformation, so that the transformation of a leaf bone is the product of the transformations of the bones from the root bone to the leaf bone. Mesh vertices are then related to bones through weights, which describe the strength of influence of a particular bone on the transformation of a vertex.

Skeleton creation can be a tedious process, usually done by hand and requiring a considerable level of expertise. The work by Wade and Parent [29] is a pioneering example of automating skeleton creation. Their approach works by converting the surface mesh to a volume representation, approximating the discrete signed distance function, and then reconstructing the discrete medial axis. Finally, the discrete medial axis is smoothed and simplified, and a graph is extracted, giving a reasonable candidate for a control skeleton. The generated skeleton can be refined by hand, which is usually a much simpler task than creating it from scratch.

For further information on skeletons, we refer the reader to the book chapter on Skeletal Structures by Biasotti et al. [4] from the book Shape Analysis and Structuring [12].

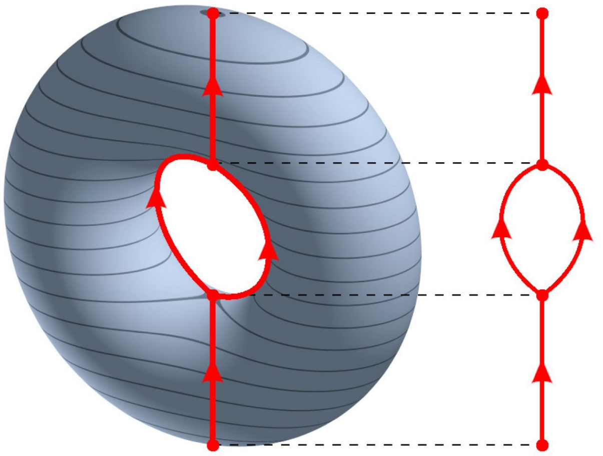
Figure 5: **Reeb graph of a torus. Source: I. Voyager, under CC PDM 1.0.**

2.3 Reeb graphs

A completely different approach is taken by **Tierny** in his PhD thesis [27], where he exploits the tools of computational topology to achieve skeletonization, and applies it to shape matching rather than animation. Morse functions are a specific class of smooth real functions defined on a manifold (usually on a surface). When the connected components of the level sets of such functions are contracted to points, and two points are connected with an edge whenever there exists a continuation between their respective level sets, the resulting graph is called a Reeb graph (see Figure 5). Such a representation forms a topological skeleton in 3D space. Reeb graphs have proven to be very flexible and robust representations of 3D shapes, especially because shape invariants can be made explicit through the underlying Morse function. As such, they have been used in many different applications, including shape matching and retrieval, surface parametrization, mesh simplification and segmentation, and animation [4].

2.4 Shock graphs

On many occasions, it may be beneficial to describe a shape (e.g. a parametrized curve or a surface) through a temporal evolution process, since we can identify and specifically represent important events, such as topological changes. Such singularities of the evolution process may be arranged in a graph that captures the essence of the shape in question. Siddiqui et al. [24] describe the 2D shape by considering a special form of a temporal evolution process from the medial axis, and the singularities that this process creates. These so-called shocks can be used to derive structural descriptors. Shock graphs represent the same topological features as medial axis transforms, but enhance them with the information related to the temporal evolution process [4]. The authors of the method presented a way to organize such shocks in a graph that can be simplified and analyzed through a specialized shock graph grammar. The grammar is in a sense a linearized description of a shape, and can be used as such for shape matching.

2.5 Motion graphs

In movies and video games, realistic motion (especially of humans) is an important requirement, contributing significantly to the perceived realism. Modern approaches usually start with motion capture data and employ machine learning to infer transformations and transitions. The work of Kovar et al. [14] is a pioneering example of extracting motion characteristics from real life capture data. The method generates convincing synthetic motion that retains the quality of the captured motion, while meeting the

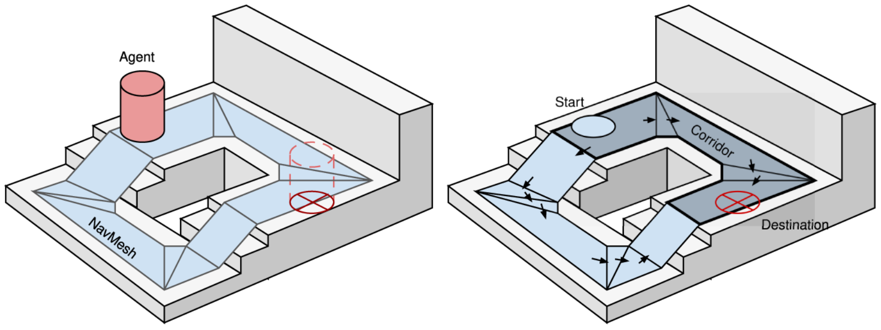

Figure 6: **An example navigation mesh (left) and the two possible paths between the start and end points. Source: Unity Editor Manual and Scripting Reference, under CC BY-NC-ND 4.0.**

user's requirements such as direction of travel and type of movement. The authors achieve this by detecting motion clips, construct a motion graph from these clips, and then use graph search techniques to find sequences that satisfy the given demands. As such, a motion graph is composed of vertices representing skeleton poses and directed edges representing the transitions (motion clips) between them. A trivial motion graph can be constructed from the captured motion by simply stitching together motion clips in a linear sequence, but this approach cannot be used to generate new motion. The authors enhance the trivial graph by adding new edges between vertices where there exists a motion clip with the corresponding start and end poses. Since it is unlikely that two poses would match perfectly, the authors allow a small error and insert a synthetic motion clip by linearly interpolating the two poses. This results in a much more complex graph with many more connections, and therefore many more synthetic motions possible. Additionally, each motion clip may be labeled with different attributes so that, during motion synthesis, pathfinding only considers motion clips including, for example, running or crouching.

2.6 Navigation graphs

Player movement in video games depends completely on the actions of the player. There is no collision avoidance, steering, or pathfinding involved. Collisions are usually handled by the physics engine, and pathfinding is left up to the player. Non-player character movement, on the other hand, requires the knowledge of walkable areas in a scene and how these areas are connected together to be able to produce convincing and apparently smart motion behavior. The navigation problem can be broken down into three parts: the first is to identify the walkable areas in a scene, the second is to break down these areas into smaller regions in which local steering behavior suffices for effective navigation, and the third is to connect the smaller regions together with connections where one can pass from one region to another. The regions of the walkable areas can be represented with vertices, which, together with the connections, form a navigation graph. Since the vertices and edges of the navigation graph require additional geometric information for local steering behavior, such as the shape and size of a region, the resulting structure is usually referred to as the navigation mesh (see Figure 6. The navigation problem is thus reduced to the shortest path problem in a navigation graph, which can be efficiently solved, for example, with the A* algorithm. Furthermore, hierarchical pathfinding can significantly reduce the time requirements by arranging the navigable areas into a hierarchy (e.g. quadtree).

Although having been automated to a large extent, navigation mesh construction remains a difficult problem and is often deferred to the artist building the scene. The work of Oliva and Pelechano [19, 20] addresses automatic navigation mesh generation. The first method [19] works in 2D, and addresses the decomposition of a polygon into convex subregions (cells) joined with line segments (portals) at intersections of adjacent cells. The subregions become vertices of a graph and the portals become edges, together forming a graph called the cell and portal graph (CPG). Navigation proceeds by employing a graph pathfinding algorithm followed by local steering behavior within a cell. Many similar techniques exist, some of them based on Delaunay triangulations or Voronoi diagrams, but the method of Oliva and Pelechano is the most common in practice because it is fast to compute and robust for use in computer games. The authors later extended the work to 3D mul-

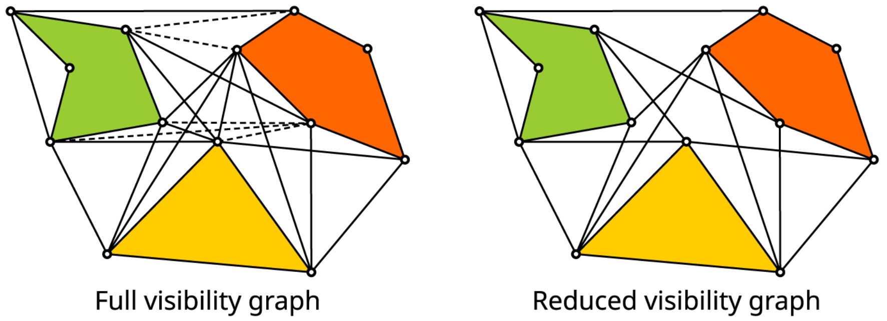

Figure 7: **Full and reduced visibility graphs between the vertices of a 2D scene. Non-tangential connections, which are considered redundant and are removed in the reduced visibility graph, are drawn with dashed lines.**

tilayered environments [20], where walkable areas of a map are automatically extracted through voxelization, then the CPG is calculated with the previous method. Similar approaches have been successfully integrated into some of the most popular game engines, such as Unity and Unreal, although manual tuning is often still required to remove potential inaccuracies.

Navigation graphs are also used extensively in robot path planning. In these scenarios, a representation of the environment must first be acquired and put into a suitable form for navigation graph generation. The work of Pütz et al. [21] uses a laser scanner mounted on top of a robot to generate a point cloud of the surrounding terrain, then converts it to a triangle mesh and generates a CPG. The CPG is further enriched with information about roughness and trafficability, which are estimated from the point cloud.

Further information on navigation meshes can be found in the comparative study by van Toll et al. [28].

2.7 Visibility graphs

The visibility graph is a fundamental geometric structure used in many different application scenarios, such as motion planning [30] and rendering optimization [5]. It is a graph of mutually visible locations in a scene, so that only the cell in which the camera is located and the adjacent cells must be rendered. In computer graphics, it is often used together with navigation graphs, which break down a scene into convex cells, in which all points are mutually visible. While navigation graphs are usually more fine-grained structures, nodes in visibility graphs tend to represent larger areas, so that even a quick visibility query can result in a large performance benefit. Nodes, which are redundant in a given context, are often removed and the resulting graph is called a reduced visibility graph (see Figure 7).

2.8 Mixture graphs

Segmentation volumes are becoming a first-class modality in many imaging scenarios, such as medical imaging, biology, histopathology, material sciences, etc. Each voxel in a segmentation volume is an integer label of a specific instance, and as such cannot be interpolated for the purposes of rendering. Although such volumes are highly compressible, queries (especially range queries) still require considerable bandwidth, which makes them impractical. While the capturing resolution and storage capacities are steadily increasing, query efficiency remains the main bottleneck in many volume-related application. Al-Thelaya et al. [1] present a method for representing segmentation volumes in a form of a directed acyclic graph (DAG) to speed up queries and at the same time efficiently compress the data. They define mixtures, specific convex combinations of segmentation labels, and organize them into DAGs, where each mixture is either explicitly stored or it is a linear interpolation of exactly two existing mixtures. Such mixture graphs can be efficiently stored and queried on a modern GPU, as demonstrated by the authors in several application scenarios.

2.9 Path graphs

Photo-realistic computer images are nowadays almost exclusively generated with a variant of the path tracing algorithm. Path tracing solves the light transport equation by shooting numerous light rays into the scene and simulating interactions between light and materials until a light source is hit. These interactions are inherently stochastic, necessitating a Monte Carlo approach. Unfortunately, the Monte Carlo method relies on the independence of samples, which means that when we start simulating the next light ray, we have to throw away all information gathered while traversing the scene, including ray-scene intersections, shadows, and shading information.

A recent publication by Deng et al. [9] describes an optimization of the path tracing algorithm that considers nearby paths through a single pixel. While bare bones path tracing independently evaluates each path through a scene and forgets all the information gathered through this process, path graphs consider clusters of at most K paths. Consequently, the rays through a single pixel do not form a tree anymore (a collection of individual paths through the scene, originating from a common location – the camera), but by adding connections between the endpoints of the neighboring paths it becomes a DAG. The resulting graph therefore contains three types of connections: light edges sampled for next-event estimation, continuation edges created through BSDF sampling to extend paths, and neighbor edges connecting to spatial neighbors within each cluster. By aggregating and propagating the radiance through these new connections, the convergence of the rendering is significantly improved, as such operations are simpler than light propagation.

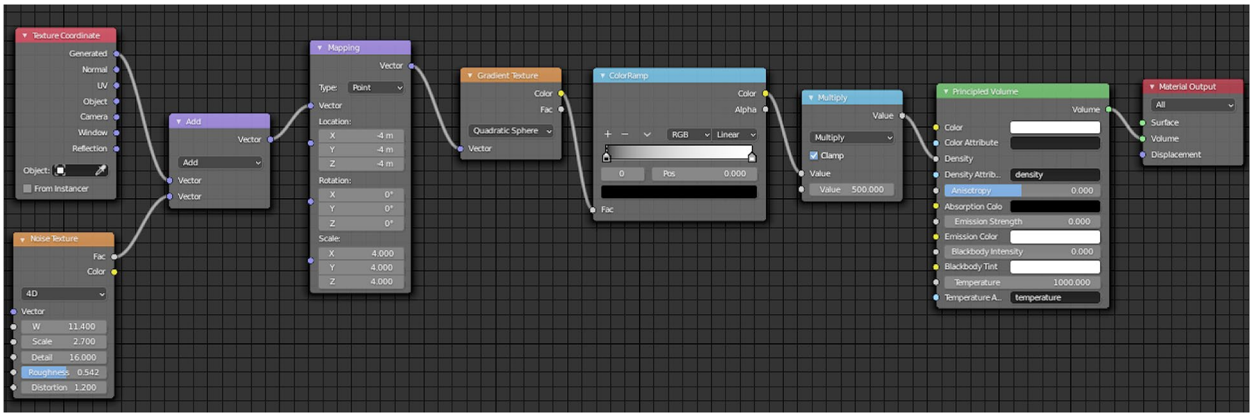

Figure 8: **An example shader graph showing a volumetric shader in Blender.**

2.10 Shader graphs

In modern graphics pipelines, the color of a pixel is defined by the shader, which outputs the final color given the material properties and illumination information. Traditionally, shaders are written in specialized languages, requiring deep knowledge about programming and GPU architectures, and as such are not suitable for artists and content creators. When describing materials, shaders can be thought of as a sequence of common operations determining how the light behaves when interacting with a given material. Such operations may include sampling from common spherical distributions and evaluating the Fresnel term. However, without any visual feedback, even such high-level computations remain too abstract for artists and content creators. Shader graphs aim to solve this problem.

Jensen et al. [13] present a system for interactive shader development in the form of a computational graph. Nodes represent operations ranging from simple multiplications to complex shading models, whereas edges are data paths connecting the output sockets of some nodes to the input sockets of other nodes. Sockets have types associated such that no invalid connections can be made. For example, if a node requires a 2D vector at a given input, the system will prevent connecting color data to it. Such a system allows for automatic code optimization techniques to be used. Furthermore, as individual computational nodes can compute relatively simple functions, it is often possible to compute the gradient as well, enabling solutions to inverse problems, such as determining material properties from photographs. Since its publication, the system has been successfully integrated into many popular game engines and 3D modeling software such as Blender (see Figure 8), Maya, and Unity.

2.11 Sparse voxel octrees

Triangle meshes have traditionally been the most common representation of surfaces because of their ease of processing and compactness for representing planar surfaces. Laine and Karras [15] note that these advantages are becoming less significant as GPU storage is becoming less of a bottleneck. They propose a hierarchical representation of surfaces with voxels, which can efficiently store geometry, shading and color information in the same data structure. In

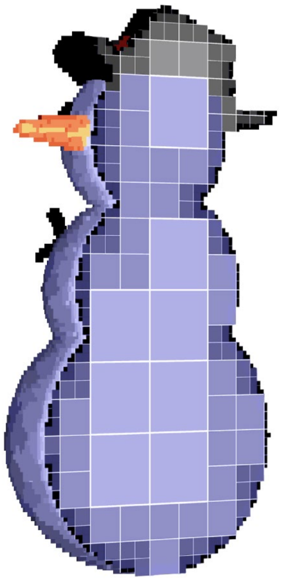

Figure 9: **An example sparse voxel octree encoding a dense 3D model. Note that this model does not store information about the actual surface, hence the blocky artifacts on the boundary. Author: Acodered, under CC BY-SA 3.0.**

a sparse voxel octree, the nodes represent a region of space, and are equipped with child occupancy flags. See Figure 9 for an example sparse voxel octree. Every node may have up to eight children, which provide additional geometric detail inside a certain subvolume. This means that accessing the geometric data of a specific point in space requires no more than O(log n) steps, where n is the number of nodes in the octree. An efficient encoding solution of the resulting sparse voxel octree is presented in the paper and showcased in a GPU ray-tracing application, where it achieves significant performance benefits compared to triangle meshes. If a less detailed model is needed, for example in streaming applications or scenarios requiring varying levels of detail, the octree can simply be pruned at a certain depth while remaining valid. After more than 10 years since publication we can safely admit that sparse voxel octrees did not replace triangle meshes, probably because of the lack of authoring tools and efficient rendering solutions with existing GPU architectures, as well as a non-trivial integration with common animation systems.

2.12 Point clouds

Point clouds are sparse representations of surfaces and volumes. They are sets of points in space with additional attributes, such as colors, surface normals, and light bounces. Point clouds are generally produced by 3D scanning devices, such as LIDAR scanners, or by photogrammetry software. Since points represent infinitely small parts of surfaces or volumes, rendering them convincingly is not trivial, but many other tasks are greatly simplified. Common tasks include efficient storage, rendering, compression, alignment and registration, and conversion to other forms of representation, usually triangle meshes. Graphs are often used to provide a general shape of the point cloud for the purposes of efficient compression of attributes and for registration.

Octrees can be used for storing point cloud data while at the same time serving as an efficient spatial data structure. Similarly to sparse voxel octrees for storing surface data, planes can be used as simple predictors of subtree occupancy, as demonstrated by Schnabel and Klein [23]. Their method works by imposing an octree structure on the point cloud, quantizing the points to the octree cell centers, and then computing the occupancy predictors with linear least squares approximation. Any differences from the original dataset are stored in the octree. Apart from point locations, the authors show that the linear least squares approximation also works extremely well with other attributes, for example colors. This approach elegantly enables progressive decoding of the point cloud when traversing it in a breadth-first manner, which is beneficial when transmitting the dataset over a network.

Most point cloud storage and compression methods focus on efficiently encoding point positions, but disregard other attributes, such as colors and normals. The above approach uses linear approximation for all attributes, which is in many cases inadequate. The method of Zhang et al. [31] specifically addresses this problem by employing a graph transform (Karhunen-Loève transform) from the field of graph signal processing to more effectively compress the data domain. The method defines a precision matrix on the graph, formed by the weighted adjacencies, then applies the transform on the matrix (assuming a Gaussian Markov random field), compacting the data and making it efficiently compressible. This is a case in which similarities in point attributes are naturally represented with connections within the graph that captures the general shape of the point cloud.

While both above methods operate on static point clouds, Thanou et al. [26] extend the compression of point clouds to point cloud sequences, where point positions and colors change smoothly over time. Such data commonly arise, for example, from RGBD video cameras, which are becoming increasingly popular, especially in video conferencing applications. As the frames usually change smoothly, the data is in theory highly compressible. Predicting the movement of the points is not trivial, though, as the number of points changes from frame to frame, and the points have no explicit correspondence between them, let alone a rigid transformation. The method works by imposing a graph structure on each point cloud so that the problem of motion estimation reduces to feature matching between successive graphs. Local features are computed with spectral graph wavelets, which facilitate feature matching. Motion is estimated on a sparse subset of the graph nodes, and the motion of other nodes is estimated through interpolation, by solving a graph-based regularization problem.

The above paper acknowledges the problem of high cost of lossless geometry coding. De Oliveira

Rente et al. [8] address this very problem with a lossy graph-based coding scheme for static point cloud geometry. The algorithm consists of two layers: a base layer, which is encoded with a scalable octree (being the most popular approach nowadays), and a graph-transform-based enhancement layer. This allows a coarser but highly compressed approximation to be displayed before the details (encoded with efficient but lossy compression) are added.

For further information on point cloud compression, we refer the reader to the survey paper by Cao et al. [6].

3 DISCUSSION

Graphs found their way into computer graphics as flexible, easy-to-implement, and robust structures for representing various kinds of data and their relations. They have been proven useful in applications such as mesh compression, animation, navigation, rendering and shape recognition. An overview of the graphs presented in this paper, along with the semantics of their constituent components, is shown in Table 1. After reviewing the publications from different fields, we see a trend in using graphs for extracting the essential information from a given context, for example topological changes in a shape, or scene connectivity, a trend which directly manifests itself in various compression schemes and efficient querying data structures. However, advanced results from network analysis seem to be rarely used, except for a few outstanding exceptions, e.g. [24, 27, 31]. Most innovation seems to reside in the connection between the actual data and the high-level graph representation, which we believe to be a vast field for further research. In particular, applications which tend to generate large and complex graphs, such as light transport simulations, would probably benefit from the approaches from network analysis.

It seems that nowadays machine learning is taking over many research fields mentioned in this work, possibly rendering the graph-based approaches obsolete. This applies to a large degree to motion synthesis, animation, material synthesis, object recognition, among others. Nevertheless, some

Table 1: **Graphs in computer graphics, their semantics, and usage.**

Graph	Vertices	Edges	Usage	References
Connectivity graph	geometric primitives	adjacencies	surface representation, mesh compression, subdivision modeling	[3, 7, 10, 11, 16, 17, 2]
Skeleton	joints	bones	animation, motion synthesis, shape matching	[29]
Reeb graph	level sets of Morse functions	level set continuation	shape matching, mesh simplification, esh segmentation, animation	[27]
Shock graph	singularities of the temporal evolution of a shape	shock adjacencies	shape matching, object recognition, computer vision	[24]
Motion graph	skeleton poses	motion clips	animation, motion synthesis	[14]
Navigation graph	simple walkable regions	connections between walkable regions	navigation,path planning	[19, 20, 21, 28]
Visibility graph	regions of space	mutual visibility between regions of space	rendering, artificial intelligence	[5, 30]
Mixture graph	subvolumes	subvolume mixtures	volume rendering, segmentation volume compression, 3D imaging	[1]
Path graph	light-material interaction locations	collision-free light paths	rendering, light transport simulation	[9]
Shader graph	shading operations	data paths	rendering, material synthesis, shader optimization	[13]
Sparse voxel octree	regions of space and enclosed surfaces	subregions and surface refinement	surface representation, rendering, level of detail	[15]
Point cloud shape graph	representative points	spatial or other similarities	point cloud compression, motion prediction, shape matching	[23, 31, 26, 8, 6]

graphs have survived for a very long time, including connectivity graphs, skeletons, and navigation graphs, and they are probably not going to be replaced anytime soon due to their effectiveness and ease of implementation. Taking advantage of both machine-learning-based and graph-based approaches could possibly revolutionize these fields or at least simplify the creation of the graphs to facilitate the workflows of content creators. Such advances are already happening in the form of mesh generation, material synthesis, automatic skeletonization, and neural-network-generated scene graphs.

4 CONCLUSION

We presented an overview of the main topics in the intersection of the fields of computer graphics and graph theory. We listed the most commonly used graphs, recent advancements and some of the most groundbreaking research. Our review revealed a trend in using graphs for extracting the essential information for the purposes of data compression, shape representation, and efficient querying. While some of the applications of graphs, especially object recognition and motion synthesis, are being slowly replaced with deep learning approaches, graphs are probably going to remain the standard workhorse in some of the important applications in video game industry, such as mesh representation and navigation.

REFERENCES

[1] Khaled Al-Thelaya, Marco Agus, and Jens Schneider. The Mixture Graph-A Data Structure for Com- pressing, Rendering, and Querying Segmentation Histograms. IEEE Transactions on Visualization and Computer Graphics, 27(2):645–655, feb 2021.

[2] Pierre Alliez and Craig Gotsman. Recent Advances in Compression of 3D Meshes. In Advances in Multiresolution for Geometric Modelling, pages 3–26. Springer-Verlag, Berlin/Heidelberg, 2005.

[3] Bruce G. Baumgart. A polyhedron representation for computer vision. In Proceedings of the May 19-22, 1975, national computer conference and exposition on – AFIPS ’75, page 589, New York, New York, USA, 1975. ACM Press.

[4] Silvia Biasotti, Dominique Attali, Jean-Daniel Boissonnat, Herbert Edelsbrunner, Gershon Elber, Michela Mortara, Gabriella Sanniti di Baja, Michela Spagnuolo, Mirela Tanase, and Remco Veltkamp. Skeletal Structures, pages 145–183. Springer Berlin Heidelberg, jan 2008.

[5] Mojtaba Nouri Bygi and Mohammad Ghodsi. 3D visibility graph. Computational Science and its Applications, 2007.

[6] Chao Cao, Marius Preda, and Titus Zaharia. 3D Point Cloud Compression. In The 24th International Conference on 3D Web Technology, pages 1–9, New York, NY, USA, jul 2019. ACM.

[7] Edwin Catmull and Jim Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. Computer-Aided Design, 10(6):350–355, nov 1978.

[8] Paulo de Oliveira Rente, Catarina Brites, Joao Ascenso, and Fernando Pereira. Graph-Based Static 3D Point Clouds Geometry Coding. IEEE Transactions on Multimedia, 21(2):284–299, feb 2019.

[9] Xi Deng, Miloš Hašan, Nathan Carr, Zexiang Xu, and Steve Marschner. Path graphs: iterative path space filtering. ACM Transactions on Graphics, 40(6):1–15, dec 2021.

[10] Daniel Doo. A subdivision algorithm for smoothing down irregularly shaped polyhederons. Computer Aided Design, pages 157–165, 1978.

[11] Daniel Doo and Malcolm Sabin. Behaviour of recursive division surfaces near extraordinary points. In Seminal graphics, pages 177–181. ACM, New York, NY, USA, jul 1998.

[12] Leila De Floriani and Michela Spagnuolo, editors. Shape Analysis and Structuring. Springer Berlin Heidelberg, 2008.

[13] Peter Dahl Ejby Jensen, Nicholas Francis, Bent Dalgaard Larsen, and Niels Jørgen Christensen. Interactive shader development. In Proceedings of the 2007 ACM SIGGRAPH symposium on Video games – Sandbox ’07, page 89, New York, New York, USA, 2007. ACM Press.

[14] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. ACM Transactions on Graphics, 21(3):473–482, jul 2002.

[15] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees. IEEE Transactions on Visualization and Computer Graphics, 17(8):1048–1059, aug 2011.

[16] Charles Loop. Smooth subdivision surfaces based on triangles. Master’s thesis, University of Utah, 1987.

[17] Charles Loop and Scott Schaefer. Approximating Catmull--Clark subdivision surfaces with bicubic patches. ACM Transactions on Graphics, 27(1):1–11, mar 2008.

[18] Bojan Mohar and Carsten Thomassen. Graphs on Surfaces. Johns Hopkins University Press, aug 2001.

[19] Ramon Oliva and Nuria Pelechano. Automatic Generation of Suboptimal NavMeshes. In *MIG’11: Proceedings of the 4th international conference on Motion in Games*, pages 328–339, 2011.

[20] Ramon Oliva and Nuria Pelechano. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. *Computers & Graphics*, 37(5):403–412, aug 2013.

[21] Sebastian Pütz, Thomas Wiemann, Jochen Sprickerhof, and Joachim Hertzberg. 3D Navigation Mesh Generation for Path Planning in Uneven Terrain. *IFAC-PapersOnLine*, 49(15):212–217, 2016.

[22] Jarek Rossignac. Edgebreaker: connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[23] Ruwen Schnabel and Reinhard Klein. Octree-Based Point--Cloud Compression. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG’06, pages 111–121, Goslar, DEU, 2006. Eurographics Association.

[24] Kaleem Siddiqi, Ali Shokoufandeh, Sven J. Dickenson, and Steven W. Zucker. Shock graphs and shape matching. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 222–229. Narosa Publishing House, 1999.

[25] Olga Sorkine and Daniel Cohen-Or. Least-squares meshes. In *Proceedings Shape Modeling Applica- tions*, pages 191–199. IEEE, 2004.

[26] Dorina Thanou, Philip A. Chou, and Pascal Frossard. Graph--Based Compression of Dynamic 3D Point Cloud Sequences. *IEEE Transactions on Image Processing*, 25(4):1765–1778, apr 2016.

[27] Julien Tierny. *Reeb graph based 3D shape modeling and applications*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2008.

[28] Wouter van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*, pages 91–100, New York, NY, USA, oct 2016. ACM.

[29] Lawson Wade and Richard E. Parent. Automated generation of control skeletons for use in animation. *The Visual Computer*, 18(2):97–110, apr 2002.

[30] Yangwei You, Caixia Cai, and Yan Wu. 3D Visibility Graph based Motion Planning and Control. In *Proceedings of the 2019 5th International Conference on Robotics and Artificial Intelligence*, pages 48–53, New York, NY, USA, nov 2019. ACM.

[31] Cha Zhang, Dinei Florencio, and Charles Loop. Point cloud attribute compression with graph transform. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 2066–2070. IEEE, oct 2014.

■

Žiga Lesar is a Ph.D. student, a researcher and a teaching assistant at the University of Ljubljana, Faculty of Computer and Information Science. He received his B.Sc. in 2014 and M.Sc. in 2018 for his work on interactive volume rendering with web technologies. His research is focused primarily on interactive computer graphics, especially volume rendering and visualization.

■

Matija Marolt is an associate professor at the University of Ljubljana, Faculty of Computer and Information Science. He received his Ph.D. in 2002 and is currently head of the Laboratory for Computer Graphics and Multimedia and is the chair for Multimedia. His research interests are in multimedia information retrieval and visualization.