Procedural Editing of Virtual Terrains Using 3D Bézier Curves

Bruno Torres do Nascimento, Cesar Tadeu Pozzer, Flavio Paulus Franzin
Universidade Federal de Santa Maria
Programa de Pós-graduação em Ciência da Computação
Santa Maria, Brazil
{*brunotn, pozzer, ffranzin*}*@inf.ufsm.br*

***Abstract*—The process of manually shaping and adding features to virtual terrains is a time-consuming task prevalent during the development process of many applications, such as games and virtual simulators. We present a technique to shape virtual terrains by creating excavations and embankments to flatten and smooth the terrain along paths defined by composite cubic 3D Bézier curves. The curves are procedurally generated based on an input set of vertices. Our approach is capable of carving smooth paths on terrains for placement or rendering of roads, railways, rivers, or any other feature that follows a path. We employ spatial hashing and data structures to optimize execution on the GPU.**

***Keywords*—Bézier curves; procedural; carving; virtual terrains, spatial hashing;**

I. Introduction

The creation of realistic and visually pleasing virtual terrains involves the addition of several types of features, which increase the overall complexity of the task. Roads, railways, and rivers are usual features that demand significant manual work to be integrated into the terrain believably, and small inconsistencies — such as an uneven road or a river that goes uphill — can be very noticeable and spoil the overall realism. These types of features also have the characteristic of following a well-defined path that can be used to fit the nearby area of the terrain to the constraints specific to that feature.

Several types of applications (e.g., games and virtual simulators) employ procedurally generated terrains created from raster elevation data (heightmaps) and vector data (points, lines, and polygons representing rivers, roads, and lakes) [1]. Procedural methods are well suited for generating large scale terrains that can be very detailed, based on the amount of input information available. One key challenge is to correctly and seamlessly integrate the terrain features — created from the vector data — into the surface of the terrain.

Heightmaps are a standard technique used to model virtual terrains [2] [3], where a scalar field of height values — usually encoded in a texture — is used to displace the vertices of a regular grid mesh, thus shaping the surface of the terrain. Heightmaps have the advantage to be easily editable, allowing the addition of local changes and offering fine control over the topography of the terrain.

As the size of the terrain and quantity of features increases, the amount of time necessary to manually edit and fit all the features can quickly become impractical, thus requiring a procedural approach. The main challenge of procedural methods is to present consistent results when faced with unforeseen scenarios.

We propose a procedural approach for integrating features into a terrain, by editing the heightmap along composite cubic 3D Bézier curves, which define the paths followed by the features, creating excavations and embankments that smoothly blend into the terrain. In order to allow making alterations in real-time with reduced overhead, our approach is designed to take advantage of the GPU for parallel processing, through the use of compute shaders. The sole purpose of our approach is to adjust the topography of the terrain to fit the features, not to handle the rendering or generation and placement of 3D models, such as roads or train tracks.

Our approach calculates, in real-time, the displacement necessary to fit the terrain, and stores the displacement values in a separated texture, thus maintaining the original heightmap values unchanged during editing. That has the advantage of allowing changes to be undone by the user so that the editing process can be more dynamic. Once the editing is finished, the user can choose to bake the displacement into the heightmap.

The main contributions of our work are:

- a technique for procedural editing heightmaps along paths defined by composite cubic 3D Bézier curves;
- an efficient method for storing and evaluating Bézier curves on the GPU;
- a straightforward algorithm to generate composite cubic 3D Bézier curves from vector data.

This paper is structured as follows: Section II explores related works. In Section III we present an overview of our approach, which is discussed more in depth in Sections IV through VII. Finally, in Sections VIII and IX, we present and discuss the results achieved.

II. Related Work

Several solutions tackle, from different perspectives, problems related to the procedural placement of paths on virtual terrains. Surveys on procedural creation of virtual worlds, presented by Smelik et al. [2], and Freiknecht and Effelsberg [3], record several works that address problems related to road generation. Smelik et al. [2] note that procedural road generation has primarily been addressed in the context of procedural cities, so the generation of interstate and country roads still requires further attention.

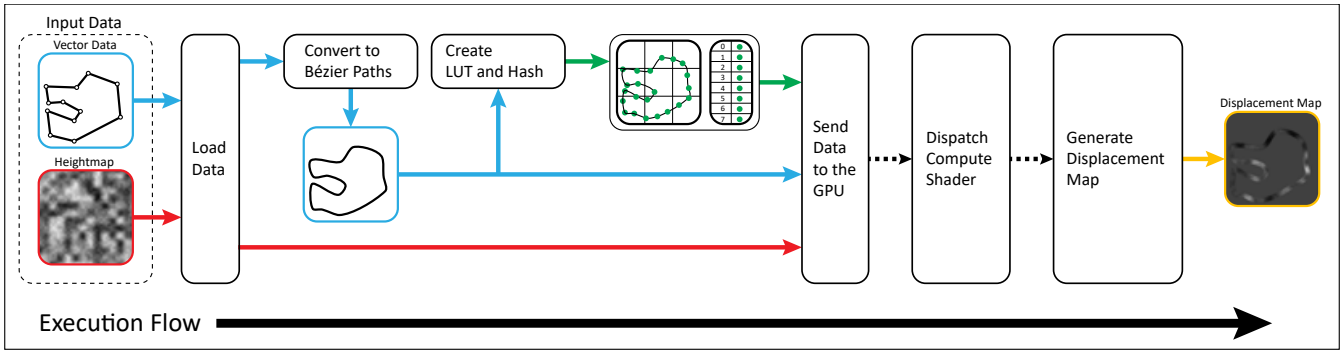


Figure 1. Overview of the execution steps. The colored arrows represent the flow of each type of data. Heightmap is red; vector data is blue; LUT and hash are green; and displacement values are yellow.

McCrae and Singh [4] present a sketch-based system for the conceptual layout of 3D path networks, where a 3D curve is created from a 2D sketch done by the user, employing a clothoid fitting approach. Although they mention the system alters the terrain to integrate the paths, no solution is presented in their paper.

Bruneton and Neyret [1] present a method to render and populate very large terrains with features. To represent road and river paths, they use 2D Bézier curves. Their approach relies on getting the curves from the input vector data and, therefore, does not provide a method for creating the curves. Furthermore, their approach relies on generating a mesh for every curve, which is used to flatten the terrain along the paths. Since the 2D curves do not encode height, they are constrained to the height of the terrain, contrary to our approach.

Works presented by Galin et al. propose methods for procedural generation of roads [5], and the generation of road networks [6]. In [5], Galin et al. focus on creating paths using a weighted pathfinding algorithm and generating procedural road models. The terrain displacement is briefly described as being performed by modifying the mesh of the terrain near the trajectory of the path, although no results illustrating the time costs of that process are provided. In [6], the approach presented by Galin et al. employs geometric graph generation combined with a path merging algorithm to create road networks between different types of roads. Terrain excavation is mentioned but not covered in that paper.

Works presented by Kelly and McCabe [7], and Parish and Müller [8] focus on procedural city modeling and road layouts; however, they do not address the terrain fitting problem. Applegate, Laycock, and Day [9] present a sketch-based system for highway design aimed at creating traffic simulations. In their approach, the terrain is displaced by constraining the height of its vertices using the road mesh.

Thöny, Billeter, and Pajarola [10] present a work focused on rendering vector features, where they employ a deferred line rendering method associated with a spatial hashing technique to improve performance when working with very large data sets. Similarly, Frasson, Engel, and Pozzer [11] present a screen-space approach for rendering vector features on virtual terrains. Their work employs an efficient data structure used to access the vector data on the GPU. Although both works achieve good rendering results, their approaches do not alter the topography of the terrain to fit the features, nor they employ Bézier curves since they are designed to work with polygons and polylines.

III. Overview

This section briefly describes all the steps our technique follows to perform the editing of the terrain heightmap (Fig. 1).

Firstly, the raster data (heightmap) and vector data (features) are loaded. The vector data consist of lists of 3D points — which we will call vertices — that describe the paths followed by the features. Optionally, the user could manually input the vertices.

The second step is to convert the lists of vertices to composite cubic 3D Bézier curves. Each pair of vertices that compose a line segment will generate a curve with four control points. The two extreme control points are the original vertices, and the other two intermediary control points will be derived based on the relative direction between adjacent line segments and their lengths. Section IV covers this process in detail.

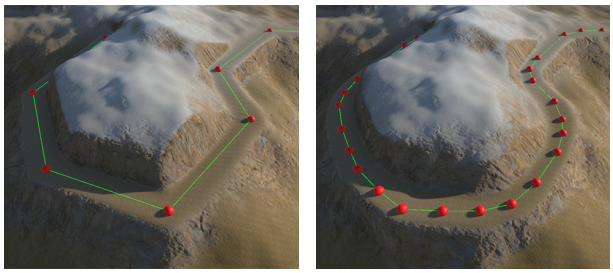
The third step is to sample the Bézier curves in small intervals to create a look-up table (LUT) that will be used to speed up queries on the GPU. To further optimize the queries, a spatial hash is created and filled with the LUT samples. This process is explained in detail in Section V.

In the fourth step, all the processed data is sent to the GPU, and a compute shader is dispatched to process the entire heightmap in parallel. Every heightmap position is sampled and subtracted from the height of the Bézier curve that passes on it — or within range. The difference values are saved in a Displacement Map and can be summed to the original heightmap values in order to get the final terrain height at each position. See Sections VI and VII for a detailed explanation.

IV. Bézier Curve Generation

As mentioned previously, the input vector data are lists of 3D vertices that represent the line segments in a path. This type of data representation usually requires a very high resolution — with more vertices, and smaller line segments — to be

able to describe smoother paths (Fig. 2). Therefore, in order to allow for easier parameterizing and editing in real-time, and to make the paths smoother, our approach is to employ composite cubic 3D Bézier curves (i.e., Bézier splines) generated from the input vector data.



(a) Low Resolution (b) High Resolution

Figure 2. Comparison between resolutions of an input path.

We chose to employ cubic curves because they offer a good tradeoff between control and computational cost, and, also, can be constructed in a manner that ensures C1 continuity between curves, without drastic changes to the shape of the path.

Each curve composing the spline is created from a line segment. Since it is a cubic spline, each curve has four control points, being the last one shared with the subsequent curve. The two vertices of a line segment become the first and last control points in the curve — henceforth called anchors —, and the two intermediate control points — henceforth called handles — are derived based on the direction and length of the adjacent line segments, and a smoothness parameter Ω. Fig. 3 illustrates — in 2D, for simplicity — the creation of a composite cubic Bézier curve with three curves from a path with three line segments.

Fig. 3.a shows an input path, with four vertices (A, B, C, D) and three line segments. The line segments are treated as vectors so that we can perform adding and scaling operations on them. For each shared vertex (B, C), we determine the direction (D1, D2) in which the handles will be placed, by adding the two adjacent line segments.

The directions are then normalized (Fig. 3.b), and, for each handle to be created, we calculate the distance from the vertex to the handle by multiplying half the length of the line segment by a smoothing factor Ω, with $\Omega \in [0, 1]$ (Fig. 3.c). Fig. 4 shows the result of different Ω values.

Then, the position of the handles (A1, A2, B1, B2, C1, and C2 in Fig. 3.d) is determined by scaling the directions D1 and D2 by the distances calculated in the previous step. Handles adjacent to vertices at the ends (A1, C2) are projected on its line segments ($\overline{AB}$, $\overline{CD}$).

Finally, we have a composite cubic Bézier curve that represents a path — we will call these curves simply Bézier paths (not to be confused with the input paths, composed of line segments). The generated Bézier path in Fig. 3.e is composed of ten control points and three Bézier curves.

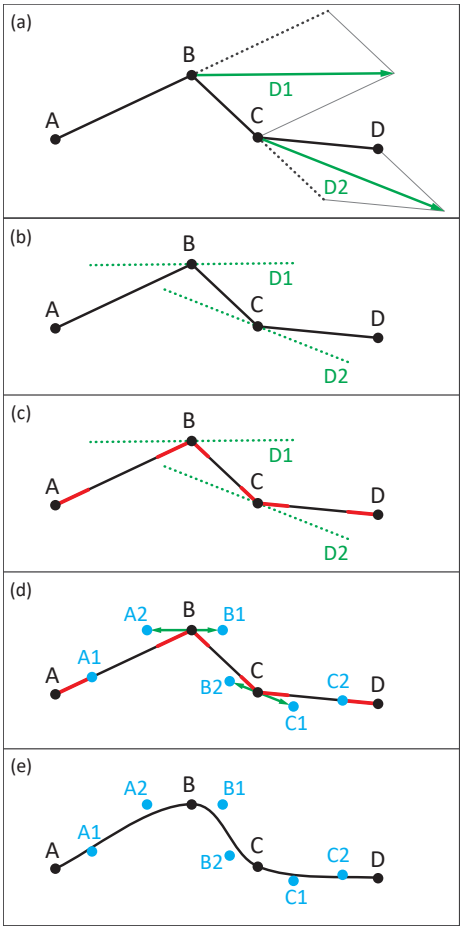


Figure 3. Generation of a composite cubic Bézier curve from three line segments. Black dots are anchors and blue dots are handles. The red line segments illustrate the distances at which the handles will be placed from the anchors when $\Omega = 0.5$.

V. Data Access and Optimization

One key aspect of our approach is, for every position on the terrain, to find the closest point among all Bézier paths that represent features to be processed. Since there is no analytical solution for point projection on cubic Bézier curves, we must resort to a numerical one. There are several approaches based on numerical methods, such as Newton-Raphson and bisection [12]–[14]. Ultimately, we chose to sacrifice precision — although the difference is visually indistinguishable — in order to reduce overhead by employing look-up tables associated with spatial hashing.

A. Look-up Table

To speed up the process of finding the closest point on a Bézier path, we sample each of its curves n times and store the results in a LUT. Each entry in the table consists of the 3D positions sampled and the parametric value t associated with the sample on the curve. Fig. 5 shows five samples (A, B, C, D, and E) obtained from a curve. Furthermore, all the curves are sampled in order, so we can calculate to which curve any

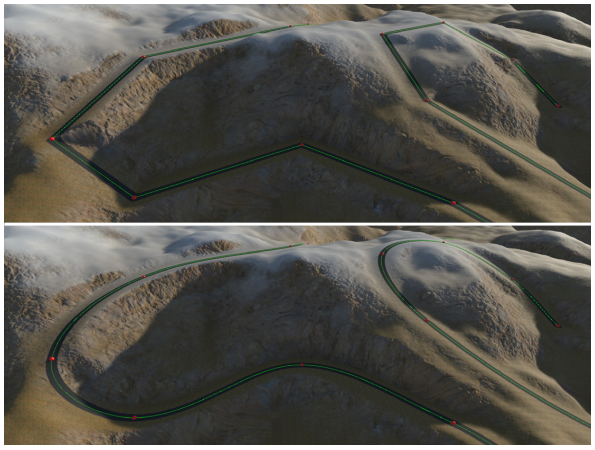


Figure 4. The smoothing factor Ω is used to reduce sharp corners. From top to bottom, Ω is 0.0, and 0.6.

entry belongs simply by dividing the index of the entry by the number of samples per curve (n). Moreover, since the last control point on a curve is shared with the next curve, we do not sample it — unless it is the last curve in a Bézier path — to avoid repeated entries in the LUT.

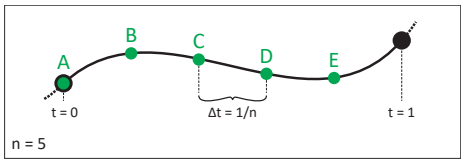


Figure 5. LUT points (in green) sampled along a curve. The black points are the anchors of the curve.

B. Spatial Hash

The LUT is very useful to find an initial approximation of the closest point on the Bézier path, however, to check the distance to every point in the LUT can quickly become too expensive. To mitigate this problem, we employ a 2D spatial hash, creating a hash table used to reduce the number of queries to the LUT. The hash table is stored in liner memory and employs an auxiliary pivot table. Its construction is based on the approach proposed by Pozzer, Pahins, and Heldal [15]. Fig. 6 shows an overview of the data organization.

Before we can construct the hash, first, we need to create and populate the LUT, as explained in the previous subsection. Then, we iterate through every point (P) in the LUT (Fig. 6.d) and, based on its (X, Z) coordinates — in this paper, we assume that the Y coordinate represents the height of the position (i.e., the Y-axis points up) —, we determine on which hash cell (Fig. 6.a) the point lies. Every entry in the pivots table (Fig. 6.b) represents one hash cell and stores how many LUT points (Fig. 6.a, in green) lie on the cell area (i.e., its usage (U)), and the starting index (I) in the hash table (Fig. 6.c).

To avoid points being incorrectly ignored when querying for distance, we employ an extra radius to determine to which hash cell they belong. Fig. 6.a shows an example of the extra radius (red circles) being utilized. Points D and F are closer to the cell boundary, and their radii overlap the neighboring cells; therefore, they are recorded as belonging to both cells. Point C, however, it is not close enough to hash cell 1, so it is recorded as belonging only to hash cell 0.

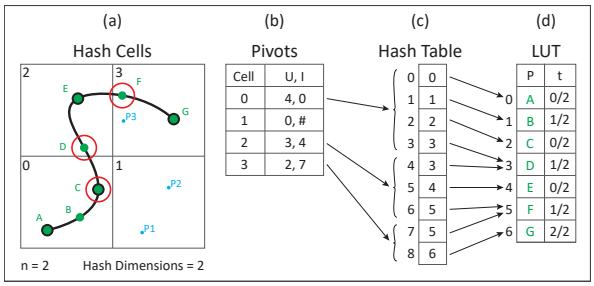


Figure 6. Data structures used to speed up queries. Hash with dimensions 2, having $2 \times 2 = 4$ hash cells. The blue points (P_1 to P_3) are examples of queried positions.

VI. Path Queries

To determine the displacement value for every position on the terrain, we need to know the point on a Bézier path that is closest to that position. We accomplish that in two stages. First, using the data structures described in previous Section, we find an approximation of the closest point on a curve. Second, we refine the result by approximating part of the curve using line segments.

A. Closest Point Approximation

When querying the closest point to a given position P, we begin by using its (X, Z) coordinates to get the hash cell where it lies. Then, we calculate the 2D distance (d) — ignoring the heights — from P to each LUT point in that hash cell to find which is the closest. In Fig. 6.a, for example, querying the position P_3 yields the LUT points F and G, and, upon calculating the distances, we find the point F as being the closest. However, querying any position in the hash cell 1 (e.g., P_1, P_2) would yield no results; therefore, we can assume that any Bézier path is too far from hash cell 1 to have any influence over it. With a LUT point in hand, we proceed to the refinement phase.

B. Closest Point Refinement

After finding the closest LUT point, we refine the result by iterating R times over the Bézier curve in a given interval of t. Fig. 7 illustrates the refinement process. The interval $[t_a, t_b]$ is determined by calculating and applying an offset μ to the value t associated with the LUT point. The value of μ is half the size of the Δt interval between LUT samples (Fig. 5) (i.e., $\mu = 0.5/n$). Consequently, $t_a = t - \mu$, and $t_b = t + \mu$. It is relevant to note that if $t_a < 0$, or $t_b > 1$, then the interval spans over two adjacent curves — which happens when the closest LUT point is also an anchor (e.g., points A, C, E, and G, in Fig. 6.a).

The next step is to sample $R+1$ points (k_1 to k_5, in Fig. 7.b) in the interval $[t_a, t_b]$. Then, the problem of finding the closest point to the queried position P becomes a trivial matter of calculating the orthogonal projection — in 2D, ignoring the heights — of P on each line segment $\overline{k_i k_{i+1}}$ and choosing the closest as the result P^*.

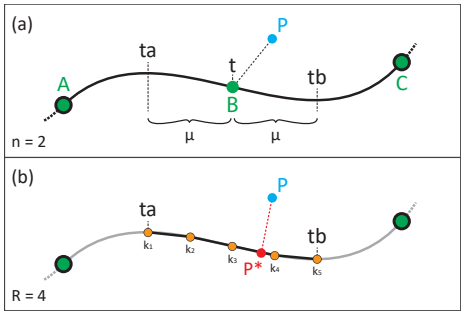


Figure 7. The two stages of querying a position P. In (a), we find the closest LUT point B and then refine the search, in (b), to get the result P^*.

VII. Displacement Map Generation

The primary purpose of the Bézier curves is to precisely describe what the terrain height should be along their path (Fig. 8). So, at every position along the path, we calculate the difference between the heightmap value (i.e., the height of the terrain) and the Y coordinate of the Bézier path (i.e., the height of the curve), and store the result in a Displacement Map. Then, when rendering the terrain, we add the heightmap and displacement values to get the final height of the terrain.

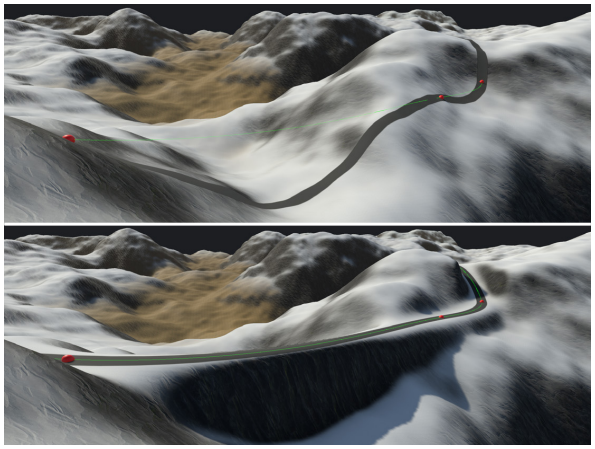


Figure 8. Terrain with no displacement (top) and full displacement (bottom) along a path.

Alternatively, the height of the curve could be baked directly into the heightmap, without the need for the Displacement Map. We understood that it is better to maintain the original heightmap unchanged, to allow for editing the paths in real-time. However, depending on the purpose, either approach could be employed.

In order to create the Displacement Map, the first step is to send all the relevant data — control points, LUT, pivot and hash tables, the number of samples per curve n and refinement steps R, and lateral smoothing parameters (w, λ; see Subsection VII-A) — to the GPU. Next, we dispatch a compute shader to process every heightmap position (i.e., texel) in parallel and save the result in the Displacement Map.

A. Lateral Smoothing

When we carve a path on the terrain, we have to consider the width w of the path. Furthermore, to create excavations and embankments, it is important to apply a smoothing distance λ to avoid sharp transitions and slopes too steep (Fig. 9). Fig. 10.a illustrates how w and λ define the area of influence of a path.

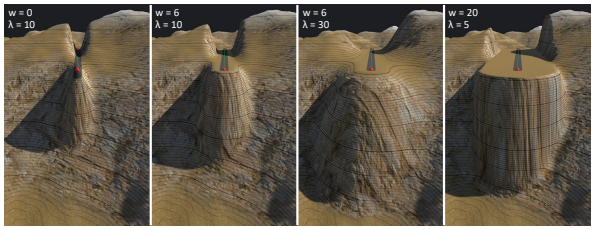


Figure 9. Excavations and embankments created with different values for w and λ.

B. Displacement Strength

To modulate the strength Θ of the displacement imprinted on the terrain by a path, first, we need to get the 2D distance d to the path — as explained in Section VI. Next, using Eq. (1), we compute the linear parameter Ψ that produces the interpolant value d, with $d \in [w/2, (w/2+\lambda)]$ (i.e., inverse linear interpolation). Fig. 10.a shows how Ψ (green line) changes across the area of influence of the path.

$$\Psi = \frac{d - (w/2 - \lambda)}{-\lambda} \tag{1}$$

To calculate Θ, we employ the 5th-order smoothstep function (Eq. (2)) proposed by Perlin [16], which has zero first and second-order derivatives at $x = 0$ and $x = 1$. This helps the displacement to blend smoothly into the terrain. Fig. 10.b shows that Θ is maximum across w and gradually decreases along λ.

$$\Theta = 6\Psi^5 - 15\Psi^4 + 10\Psi^3 \tag{2}$$

C. Displacement Calculation

The displacement calculations are executed on the GPU, in parallel, by dispatching a compute shader that runs one thread per heightmap texel. Each thread, first, converts the texel position to a world position and uses that to query the closest point P^* on a Bézier path. Next, P^* is used to calculate Θ, and the terrain height is sampled from the heightmap and subtracted from P^* height to produce a raw displacement value Δh. Finally, Δh is multiplied by Θ, resulting in the final displacement value, which is then stored in the Displacement Map.

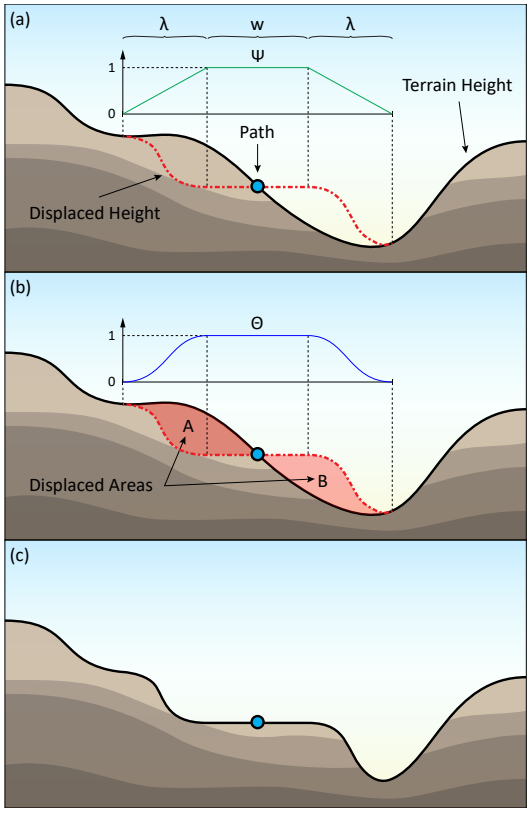


Figure 10. Cross-section of the terrain showing the lateral influence of a path, defined by the width w and the smoothing distance λ. The red dotted line in (a) represents terrain height after displacement. In (b), A and B are the areas that will be carved and filled, respectively. In (c), we can observe the shape of the terrain after the displacement.

VIII. Results

All the experiments and measurements were performed on an Intel Core i7-4790 3.5 GHz processor, with 24 GB of DDR3 RAM and a NVIDIA GeForce GTX 1070 graphics card, with 8 GB DDR5 VRAM.

A. Visual Analysis

To evaluate the visual aspects of our approach, we implemented a virtual terrain renderer that employs a vertex shader to displace a regular grid mesh using an input heightmap, which is generated using Perlin Noise [17]. In Fig. 11, we can observe a Displacement Map created from a Bézier path and used to carve the terrain. In that example, both maps have a resolution of 512x512. In this paper, all tests were performed using heightmaps and Displacement Maps that share the same resolution, although, this is not a requirement.

Using a Displacement Map with a lower resolution can cause artifacts unless the application of the maps is restricted to a part of the heightmap with the same dimensions. That can be useful in a situation where only a portion of the terrain needs to be displaced.

Fig. 12 shows a river path created using different values for Ω. Lower values can be used when it is necessary to maintain a higher fidelity to the original shape of the input path. However, we can observe that even a maximum value (bottom image) does not drastically alters the path.

The footprint of the path is parameterized and can be modified to create different effects. Fig. 9 illustrates how w and λ are used to change the profile of a path and more easily blend it in the terrain. Furthermore, we can observe, in Fig. 13, that our approach correctly displaces the terrain along the path, smoothly blending the changes to the surroundings.

Our approach can effectively be used to level the terrain along a path, allowing for rendering roads directly on the terrain, or placing 3D models, such as a railway, or water mesh, as exemplified in Fig. 14. See the attached video[1] for more results.

B. Performance

We executed several measurements in order to evaluate the performance of our approach regarding the time to generate the Displacement Map. Every measurement used the same input vector data (Fig. 15). Also, the terrain size (in world space) was 2048×2048, and the extra radius (see Subsection V-B) was 30.

To better identify the contribution of each parameter used in the generation process, we conducted the measurements by varying each value individually. Fig. 16 presents the results, grouped by parameter. All maps are square, so a resolution of 1024 means $1024 \times 1024 = 1{,}048{,}576$ texels (i.e., positions).

As expected, analyzing the results in Fig. 16, we verify that the heightmap resolution carries the most significant weight, since it directly determines the number of positions processed. Furthermore, we observe that the number of samples per curve also profoundly affects the generation times. That was worsened by the fact that the hash was not used, meaning all LUT points were accessed for every position processed.

To assess the performance gain caused by employing the spatial hash, we executed measurements using different hash dimensions. The hash is bidimensional, so a hash dimension value of 4 means $4 \times 4 = 16$ hash cells. Fig. 17 shows the results obtained, demonstrating the positive impact of the hash, which made the Displacement Map generation 8 to 63 times faster, in that scenario.

IX. Conclusion and Future Work

We have presented a technique able to procedurally carve 3D paths on virtual terrains, reducing the need for manual work editing heightmaps. Also, our approach is capable of creating composite cubic 3D Bézier curves from an input set of vertices, with smoothness parameterization. The footprint of the paths is parameterized and properly blends into the terrain. The data structures and spatial hashing presented proved to successfully reduce the overhead associated with path queries performed on the GPU. Also, our approach for approximating and evaluating Bézier curves on the GPU produced high-grade

[1] https://youtu.be/p_NPN3WVPjk

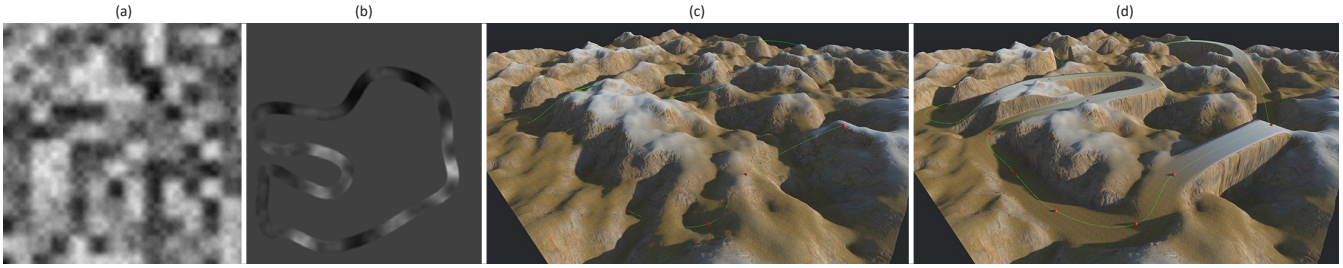


Figure 11. Example of an input heightmap (a), a Displacement Map (b), and a terrain rendered without (c) and with (d) the Displacement Map. In (b), positive and negative displacements (i.e., embankments and excavations) are represented by the lighter and darker values, respectively.

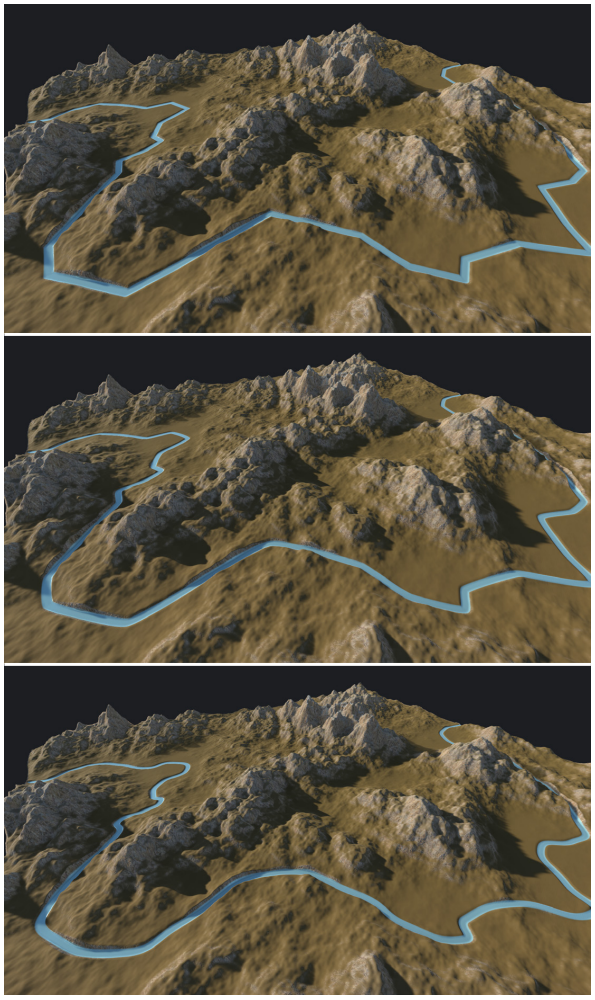


Figure 12. Comparison between different smoothing factors Ω used for a river path. From top to bottom, Ω is 0.0, 0.3, and 1.0.

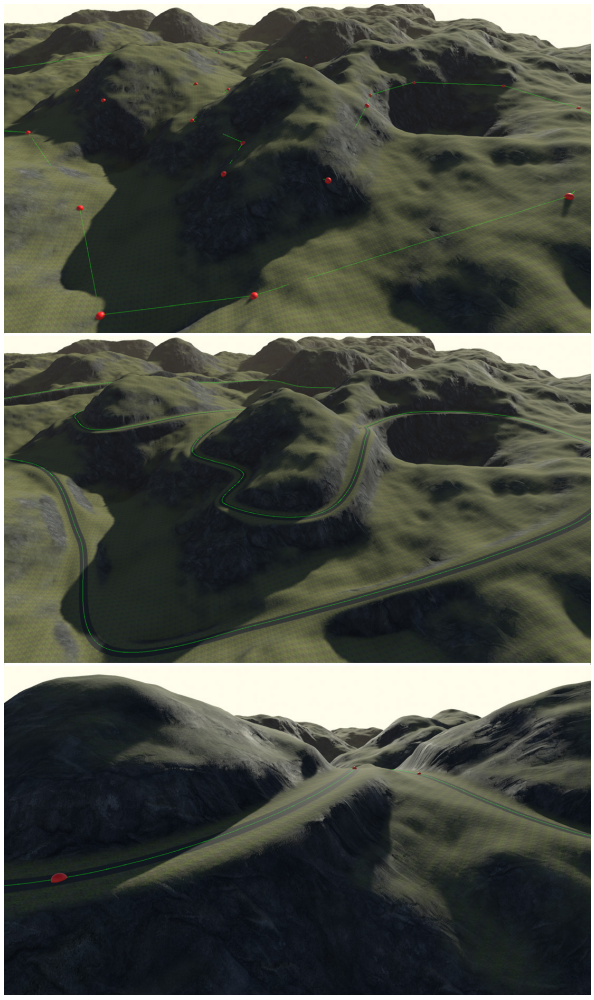


Figure 13. In the top, the original terrain and the input vector data vertices (red dots). In the middle, the Bézier path created and carved on the terrain. The bottom image shows part of the carved path in detail.

visual results, with seemingly no precision issues. Furthermore, the results presented indicate that the Displacement Map generation is compatible with real-time execution.

Currently, our approach does not handle correctly crossing or overlapping paths. However, if the paths intersect (i.e., have the same height at the crossing points), that problem is reduced. Another drawback is that, if the terrain is very large and the paths cover only a small portion of it, most of the Displacement Map will remain unused, causing a waste of memory. That problem can be mitigated by employing a Displacement Map atlas, where each page of the atlas is associated with only a small portion of the terrain. Moreover, the paths are carved using a constant footprint across its

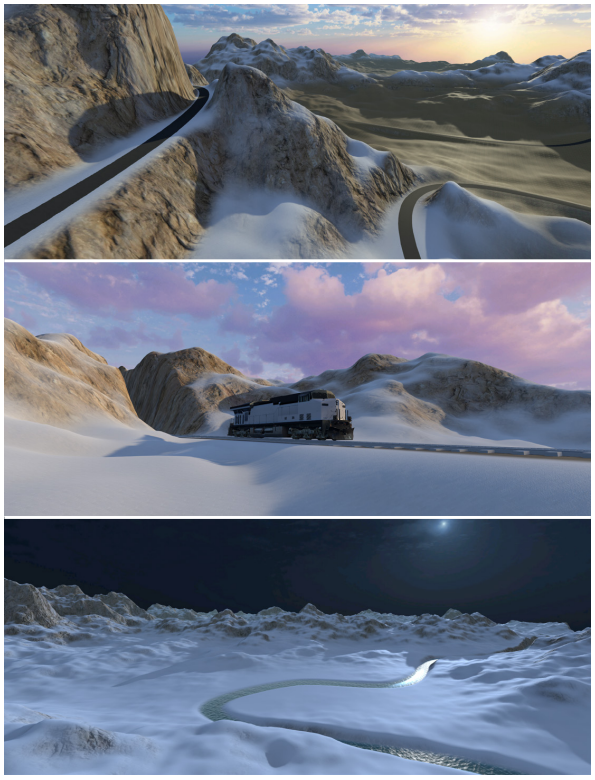


Figure 14. Roads (top), railway (middle), and river (bottom) paths carved on a terrain.

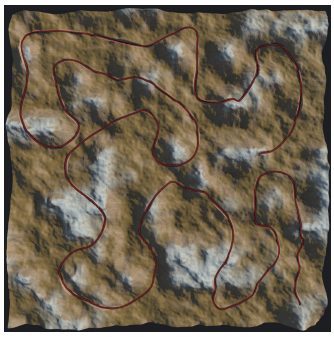


Figure 15. Bézier path (in red) used for all the measurements. The curve was created from an input vector data set containing 100 vertices.

extension, which could be improved by associating different values of w and λ to each control point.

Beyond that, we intend to associate a normal vector to the control points, so we can manipulate the inclination of the path, allowing for more realistic roads. Furthermore, it would be interesting to add support to process polygon vector data, so we could use that to carve lakes and large rivers, as well as level urban areas, among other types of features. Moreover, further optimization can be achieved by processing small sections of a path at a time, favoring real-time editing.

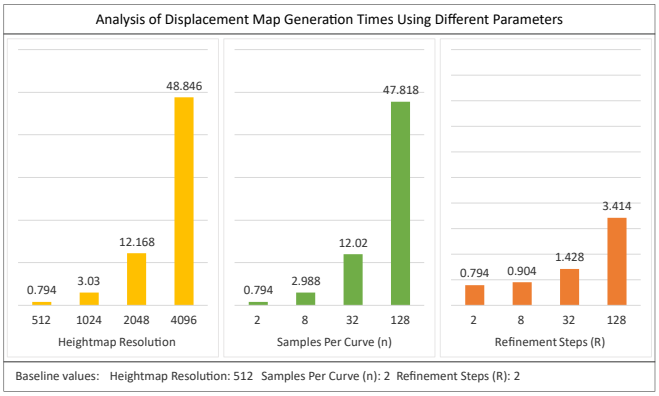


Figure 16. All times are in milliseconds. Each chart presents the results of a group of measurements performed by changing only one parameter. The remaining parameters maintain the baseline values. No hash was employed in these tests.

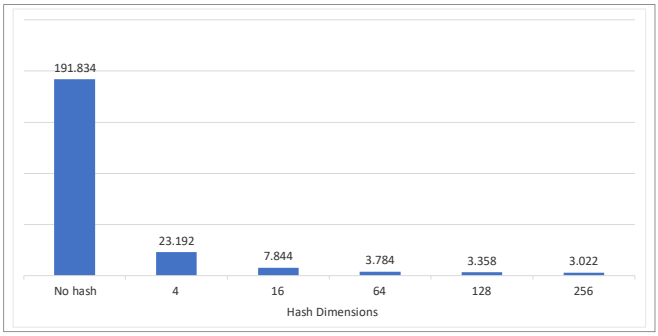


Figure 17. All times are in milliseconds. The hash dimensions determine the number of hash cells. Heightmap resolution, samples per curve, and refinement steps were, respectively, 2048, 32, and 32.

Acknowledgment

We thank the Brazilian Army for the financial support through the SIS-ASTROS project.

References

[1] E. Bruneton and F. Neyret, "Real-time rendering and editing of vector-based terrains," in *Computer Graphics Forum*, vol. 27, no. 2. Wiley Online Library, 2008, pp. 311–320.

[2] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds," in *Computer Graphics Forum*, vol. 33, no. 6. Wiley Online Library, 2014, pp. 31–50.

[3] J. Freiknecht and W. Effelsberg, "A survey on the procedural generation of virtual worlds," *Multimodal Technologies and Interaction*, vol. 1, no. 4, p. 27, 2017.

[4] J. McCrae and K. Singh, "Sketch-based path design," in *Proceedings of Graphics Interface 2009*, ser. GI 2009. Toronto, Ontario, Canada: Canadian Human-Computer Communications Society, 2009, pp. 95–102.

[5] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin, "Procedural generation of roads," in *Computer Graphics Forum*, vol. 29, no. 2. Wiley Online Library, 2010, pp. 429–438.

[6] E. Galin, A. Peytavie, E. Guérin, and B. Beneš, "Authoring hierarchical road networks," in *Computer Graphics Forum*, vol. 30, no. 7. Wiley Online Library, 2011, pp. 2021–2030.

[7] G. Kelly and H. McCabe, "Citygen: An interactive system for procedural city generation," in *Fifth International Conference on Game Design and Technology*, 2007, pp. 8–16.

[8] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 301–308.

[9] C. S. Applegate, S. D. Laycock, and A. Day, "A sketch-based system for highway design," in *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling*. ACM, 2011, pp. 55–62.

[10] M. Thöny, M. Billeter, and R. Pajarola, "Deferred vector map visualization," in *SIGGRAPH ASIA 2016 Symposium on Visualization*. ACM, 2016, p. 16.

[11] A. Frasson, T. A. Engel, and C. T. Pozzer, "Efficient screen-space rendering of vector features on virtual terrains," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2018, p. 7.

[12] X.-D. Chen, Y. Zhou, Z. Shu, H. Su, and J.-C. Paul, "Improved algebraic algorithm on point projection for bézier curves," in *Second International Multi-Symposiums on Computer and Computational Sciences (IMSCCS 2007)*. IEEE, 2007, pp. 158–163.

[13] L. A. Piegl and W. Tiller, "Parametrization for surface fitting in reverse engineering," *Computer-Aided Design*, vol. 33, no. 8, pp. 593–603, 2001.

[14] W. Boehm, "Inserting new knots into b-spline curves," *Computer-Aided Design*, vol. 12, no. 4, pp. 199–201, 1980.

[15] C. T. Pozzer, C. A. de Lara Pahins, and I. Heldal, "A hash table construction algorithm for spatial hashing based on linear memory," in *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*. ACM, 2014, p. 35.

[16] K. Perlin, "Improving noise," in *ACM transactions on graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 681–682.

[17] ——, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.