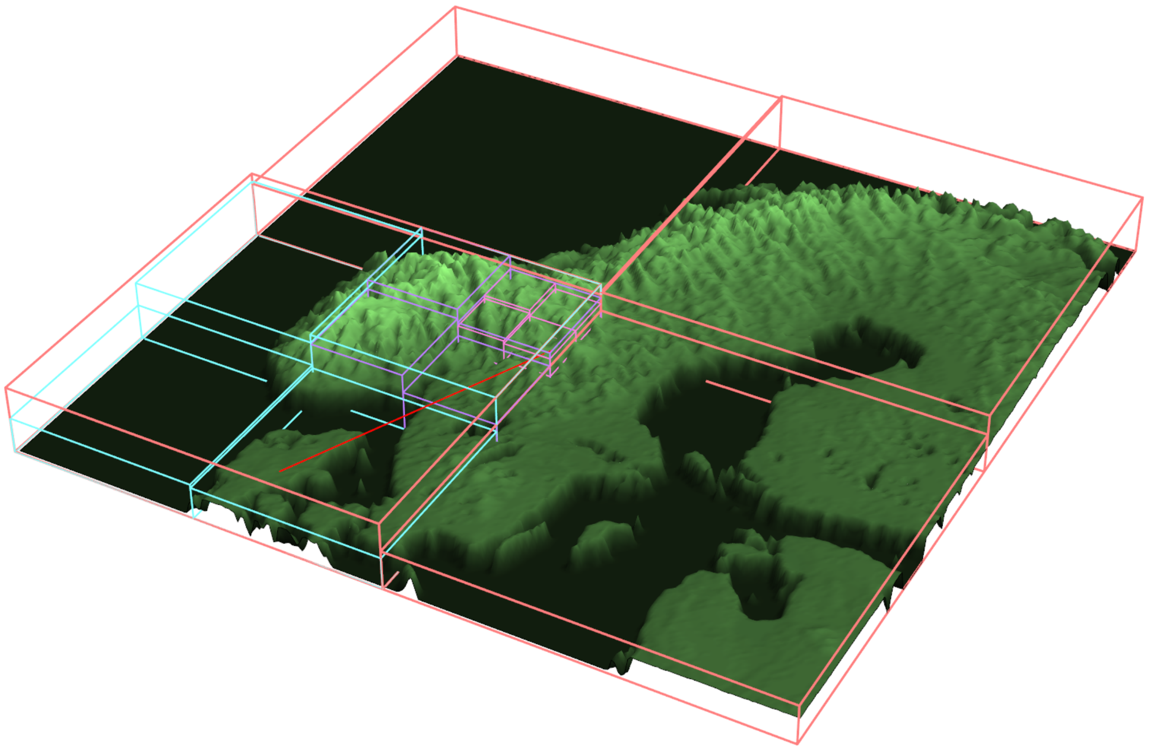
CHALMERS | UNIVERSITY OF GOTHENBURG



Efficient Intersection of Terrain Geometry in Real-Time Applications

Bachelor's thesis in Computer Science and Engineering

TIM SJÖSTRAND

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2017

BACHELOR'S THESIS

Efficient Intersection of Terrain Geometry in Real-Time Applications

TIM SJÖSTRAND

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2017

Efficient Intersection of Terrain Geometry in Real-Time Applications
TIM SJÖSTRAND

Examiner: Peter Lundin

ISSN 1654-4676

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
Visualization of investigated quadtree nodes when a ray (red line) traverses some terrain geometry.

Chalmers Reproservice
Göteborg, Sweden 2017

Efficient Intersection of Terrain Geometry in Real-Time Applications

TIM SJÖSTRAND
Department of Computer Science and Engineering, Chalmers University of Technology

University of Gothenburg
Bachelor's thesis

Abstract

In interactive computer graphics applications it is often desirable to draw some form of terrain; ground planes with visually interesting features such as valleys and hills. These are typically stored as a height map image file and converted into a collection of triangles at run-time. When such geometry is used it soon becomes necessary to calculate the intersection point of a ray through the geometry, most notably to map user cursor interaction into 3D space. However, such calculations scale poorly with the number of triangles generated from very large height maps. The traditional optimization has been to reduce the number of calculations required by utilizing a quadtree data structure to search the height map and discard triangles that do not lie inside quadtree leaf nodes along the path of the ray. This paper introduces an extension of this optimization by projecting the line through quad tree leaf nodes onto the height map plane, reducing the number of calculations by a considerable amount. Our testing of a worst-case scenario suggests improvements in calculation speed of about 50x compared to the traditional optimization, which should be interesting for anyone implementing 3D terrain based on a height map.

Keywords: 3D Graphics, Intersection Algorithm, Terrain, Ray, Heightmap, Quadtree

Acknowledgements

I would like to thank Ulf Assarsson at Chalmers for supervising this thesis as well as John Wordsworth, Johan Yngman and Marcus Beausang at Paradox Development Studio for their invaluable assistance.

Nomenclature

CPU	Central Processing Unit of a computer
GPU	Graphics Processing Unit of a computer
Shader	Program run in parallel on the GPU
AABB	Axis-Aligned Bounding Box

Contents

1 Introduction

1.1 Problem area

In modern computer graphics applications and especially real-time 3D games there is often a need to render terrain; some visually interesting ground plane with varying altitudes such as hills, mountains and valleys.

The de facto method for rendering such terrain is to generate a 3-dimensional geometry mesh from a so called *height map*; a 2D matrix of arbitrary dimension where each element denotes a height value. This height map is commonly stored as an image file, which enables the creator to conveniently edit the height map in standard image editing software. When a graphics application starts the height map is loaded and converted into 3D geometry by connecting the vertices of a set triangles to 3 corresponding elements in the height map. The vertex coordinates are calculated by sampling a point $\{x, y\}$ in the matrix, which is used as the value on the Z-axis for that vertex. The result is a highly detailed mesh of triangles that can scale in detail according to the dimension of the height map.

In recent years the emergence of *vertex shaders*, programs specifically designed to generate 3D geometry in parallel on a GPU, has solidified this method due to how vertex shaders take images as input and are able to dynamically adjust the level of detail of the resulting terrain mesh in real-time.

When intersection is to be performed on such terrain geometry (e.g. determining where to place an object without it clipping below the terrain) there are a large number of triangles to consider. When using vertex shaders, the triangles may not even exist in the memory of the CPU doing the intersection calculation as they exist solely on the GPU. Many authors, including Möller and Trumbore [2], have demonstrated how to efficiently perform ray-triangle intersection. In the case of highly detailed terrain geometry, however, the number of triangles to consider quickly grows to become computationally impractical even for fast intersection algorithms. This paper will demonstrate how it is possible to prune the majority of triangles needed for consideration by utilizing basic geometry intersection algorithms coupled with knowledge of the precise structure of the terrain geometry.

1.2 Purpose

The purpose of this paper is to explore an efficient algorithm for determining the intersection point of a ray and height map geometry in real-time. The theory is that by combining several well known algorithms for intersecting geometric primitives coupled with knowledge of the structure of the terrain geometry, the number of triangles that need to be tested in the geometry can be significantly reduced.

1.3 Limitations

The algorithm will be limited to:

- Strictly testing for rays through the terrain geometry. Support for other primitive intersections may be possible, but are not covered in this paper.
- Returning the first intersection point only. Any subsequent intersection is ignored to save time.
- The input height map matrix must have dimensions that are some number on the form of 2^n where $n \in Z^+$, but the dimensions do not have to be square.

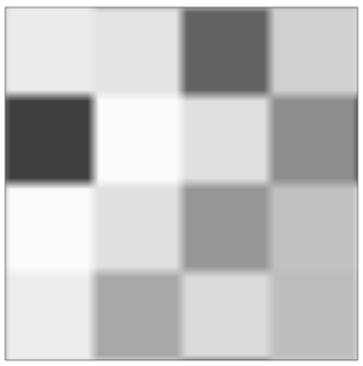
2 Technical background

The following sections will describe the key data structures and algorithms used in the newly proposed method.

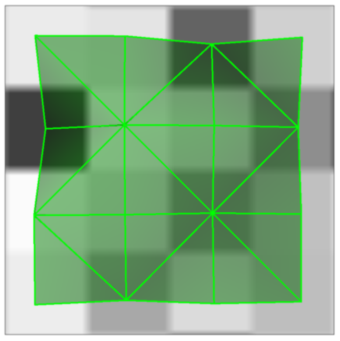
2.1 Height map

A height map is a 2D matrix of dimensions $n \times m$ where every element describes the height value for an $\{x, y\}$ coordinate in some 3D geometry mesh.

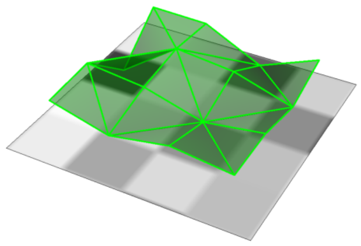
There are a number of approaches to creating geometry from a height map depending on what properties are desired for the resulting geometry mesh. For the purpose of this paper, a method suitable for terrain geometry with hills and valleys is used: first, three neighbouring points in the height map are sampled and form the vertices in a right triangle. This triangle has its 90 degree angle in the bottom-left corner, and the next triangle will have it in the upper right corner. These two triangles now form a rectangle. For the next rectangle, the process is repeated but the right angles are mirrored along the horizontal axis (see Figure 2.1). This pattern then alternates with every other rectangle.



(a) *Height map values visualized as grayscale colors (darker is lower height value).*



(b) *Triangles generated from the height map; every vertex maps onto an element in the height map.*



(c) *Perspective view.*

Figure 2.1: *Process of geometry generation from a height map; notice the repeating pattern from the bottom left set of 4 triangles in (b) and forward.*

Using this method, the number of triangles required for a $n \times m$ matrix is:

$$2(n-1)(m-1)$$

2.2 Rays

In this context a ray is simply a 3-dimensional line with an origin point and a direction vector that extends infinitely:

```
struct ray {
    vec3 origin;
    vec3 direction;
};
```

With the parametric equation:

$$P = ray_{origin} + t \times ray_{direction}$$

2.3 Axis aligned bounding box (AABB)

The axis aligned bounding box (AABB) is a subset of the minimum bounding cuboid for a (or a set of) geometry primitive(s), with the restriction that the AABB must lie in parallel with the coordinate system axes. The AABB is in general a useful structure for approximating the bounding volume around a set of primitives, and since the assumption is that the box is parallel to the axes, the intersection computation for a ray and an AABB can be made to be inexpensive.

Conventionally AABBs are represented using a minimum and maximum point, from which the other vertices of the box can be extrapolated:

```
struct aabb {
    vec3 min;
    vec3 max;
};
```

2.4 Quadtree structure

To reduce the number of intersection tests required, we introduce a quadtree data structure to efficiently search the 2-dimensional height map. The quadtree allows us to partition the full height map into quadruples of sequentially smaller areas. In quadtree terminology these areas are known as *nodes* and at the lowest level they are *leaf nodes*. In particular, the variety of quadtree sometimes known as tree-pyramid or T-pyramid where every leaf node is found at the same level is used here due to how it can be implemented efficiently in source code by using an array data type (similarly to binary trees).

The required subdivisions of a T-pyramid quadtree of size $n \times n$ into leaf nodes of $l \times l$ where n and l are powers of 2 is:

$$log_2(\frac{n}{l}) \tag{2.1}$$

and the total number of nodes required for such a tree is:

$$\sum_{m=0}^{log_2(n/l)} 4^m \tag{2.2}$$

As seen in Figure 2.2, computing a quadtree with small leaf nodes for large matrices comes at a heavy cost of memory resources - even before taking into account that every node will be represented by multiple bytes each.

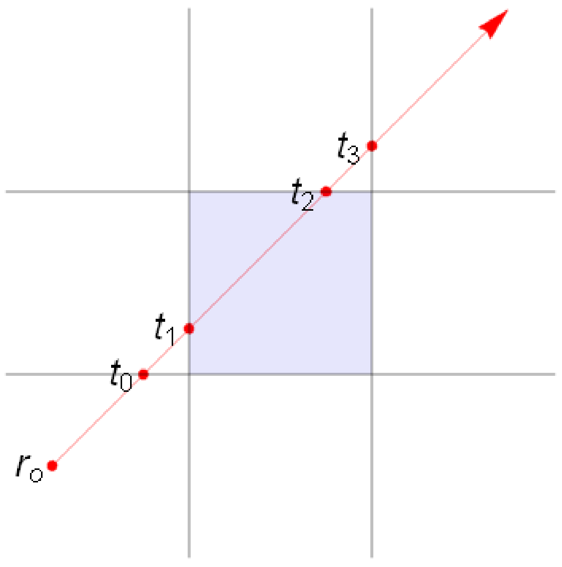
Since each node is subdivided into 4 other nodes until they reach the leaf node size, the choice of that size is key to achieving good a performance/memory ratio when utilizing quadtrees. The actual leaf node size will depend on the particular application and the dimensions of the height map matrix, where the least memory conservative would be leafs of 1×1 elements. For matrices with large dimensions however, such as the ones targeted with this algorithm, larger leaf nodes will be required.

Leaf node size	Subdivisions	Nodes total
1×1	13	89478484
8×8	10	1398100
16×16	9	349524
32×32	8	87380
64×64	7	21844

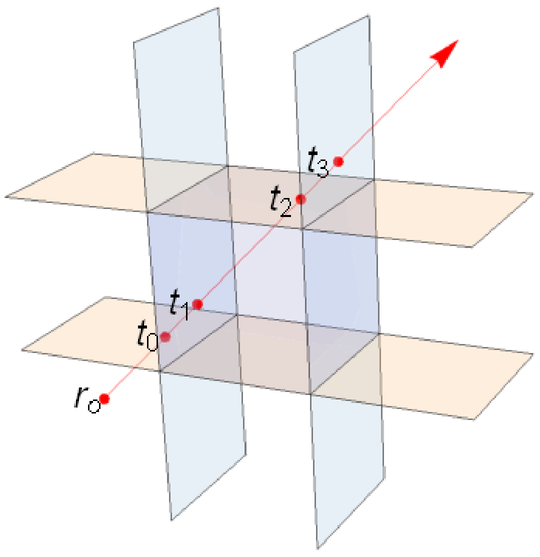
Figure 2.2: *Comparison of the total number of nodes required for a height map matrix of* 8192×8192 *using different leaf node sizes, calculated from the formulae in 2.1, 2.2.*

2.5 Ray-AABB intersection

As noted by Smits [3] and then implemented by Williams et al. [4], computationally and memory efficient ray-AABB intersection is possible by utilizing the *slab method*. With this method, the AABB is conceptually surrounded by pairs of parallel planes and the ray is clipped against each pair (see Figure 2.3). If any portion of the ray still exists after clipping, the ray intersected the AABB.



(a) *Front*



(b) *Perspective*

Figure 2.3: *Slab intersection of a ray (red) through an AABB (purple): here* t_1 *and* t_2 *are the intersection points* t_{near} *and* t_{far}*, respectively.*

In addition, this method provides us with t_{near} and t_{far}, the distance to the nearest and furthest plane intersection. These distances can be used to compute the entry and exit points of the ray through the AABB, which will prove useful later to prune additional possible triangle intersection tests.

2.6 Ray-Triangle intersection

The algorithm suggested by Möller and Trumbore [2] is appropriate for this application due to it's favorable speed and minimum precomputation requirement. The implementation is very elegant, requiring only a handful of lines of source code, but the mathematical proof is quite intricate and will not be expanded upon in this paper (see the original paper). Suffice it to say, it works and with only a few vector multiplications and scalar divisions.

2.7 Line rasterization

In 1965 Bresenham [1] outlines the line algorithm now ubiquitous for digital rasterization of lines in computer graphics. While only giving an approximation of the discrete points on a line it is well regarded for not requiring any divisions or floating point capabilities, and thus can be made to run very fast on modern computer hardware which is particularly important in a computer graphics context.

3 Implementation

The following sections will describe the steps required to implement a method that takes a ray and a height map as input and outputs the first intersection point.

3.1 Building the quadtree

First, a quadtree representing the height map is precomputed and cached for subsequent searches.

When building the quadtree we create the required nodes with their area as well the maximum and minimum height value for all the height map elements included in that area.

```
struct rect {
    vec2     min;
    vec2     max;
};

struct quadtree_node {
    struct rect     area;
    float           height_min;
    float           height_max;
};
```

Using this information, we can later construct an AABB for the node. Since Ray-AABB intersection is trivial [3] [4], we can now efficiently query the quadtree for all the leaf nodes intersected by a ray and be certain that *if* a triangle is intersected, it must be inside one of these leaf nodes.

Since the quadtree is a T-pyramid it can be implemented with a backing array similarly to a binary tree:

```
Require: $heightmap\_size \geq 1, leaf\_size \geq 1$
  $d \leftarrow log_2(heightmap\_size/leaf\_size)$ {Required subdivisions}
  $nodes\_count \leftarrow \sum_{m=0}^{d} 4^m$
  $nodes \leftarrow$ array of $nodes\_count$ nodes
  $node \leftarrow nodes_0$
  $node_{area.min} \leftarrow \{0, 0\}$
  $node_{area.max} \leftarrow \{heightmap\_size, heightmap\_size\}$
  for $i = 1; i < nodes\_count; i = i + 4$ do
    $parent \leftarrow nodes_{(i-1)/4}$
    $node \leftarrow nodes_i$
    $node_{area} \leftarrow NorthWestQuadrant(parent_{area})$
    $node \leftarrow nodes_{i+1}$
    $node_{area} \leftarrow NorthEastQuadrant(parent_{area})$
    $node \leftarrow nodes_{i+2}$
    $node_{area} \leftarrow SouthWestQuadrant(parent_{area})$
    $node \leftarrow nodes_{i+3}$
    $node_{area} \leftarrow SouthEastQuadrant(parent_{area})$
  end for
```

After the tree has been created the height values for all its nodes are calculated. To reduce the number of calculations required, this is done in reverse order; leaf nodes sample the height map, and parent nodes sample their children recursively.

```
$leaf\_first\_index \leftarrow \sum_{m=1}^{d-1} 4^m$
for $i = nodes\_count - 1; i \geq 0; i = i - 1$ do
    $node \leftarrow nodes_i$
    $height_{min} \leftarrow +\infty$
    $height_{max} \leftarrow -\infty$
    if $i \geq leaf\_first\_index$ then
        for all $\{x, y\}$ in $node_{area}$ do
            $value \leftarrow HeightmapValue(x, y)$
            $height_{min} \leftarrow min(value, height_{min})$
            $height_{max} \leftarrow max(value, height_{max})$
        end for
    else
        for $c = (i * 4) + 1, max = c + 4; c < max; c = c + 1$ do
            $child = nodes_c$
            $height_{min} \leftarrow min(child_{height\_min}, height_{min})$
            $height_{max} \leftarrow max(child_{height\_max}, height_{max})$
        end for
    end if
    $node_{height\_min} = height_{min}$
    $node_{height\_max} = height_{max}$
end for
```

3.2 Searching the quadtree

Beginning with the first node in the quadtree, we check for an intersection with the ray. If there is an intersection, that node is subdivided into it's 4 child nodes that also are tested for intersection. If any intersected node is also a leaf node, we store a result record with a reference index of the node as well as the distance to the entry (t_{min}) and exit distances (t_{max}) into the node AABB:

```
struct quadtree_intersection {
    int      node_index;
    float    t_min;
    float    t_max;
};
```

The actual entry and exit point is calculated by plugging in t_{min} or t_{max} into the parametric equation of the ray.

```
$ray \leftarrow \{origin, direction\}$
$queue \leftarrow nodes_0$
$intersected \leftarrow \{\}$
while $queue$ not empty do
    $node \leftarrow dequeue(queue)$
    $\{intersected, t_{min}, t_{far}\} \leftarrow RayVsAABB(ray, node)$
    if $intersected =$ true then
        if $node$ is $leaf$ then
            $intersected \leftarrow intersected \cup \{node_{index}, t_{min}, t_{far}\}$
        else
            $queue \leftarrow queue \cup subnodes(node)$
        end if
    end if
end while
return $intersected$
```

3.3 Ordering intersected leaf nodes

To guarantee that the first intersection found is the one closest to the ray origin, the result records are sorted in ascending order with respect to t_{min}.

For small leaf node sizes, it may be sufficiently fast to test the triangles contained in the intersected nodes at this stage. For large leaf nodes however, we can prune more potential tests.

3.4 Intra-leaf pruning

Given that:

1. Each leaf node represents a 2-dimensional submatrix in the height map, and:
2. We are testing a ray (essentially, a line) through this rectangle;

The Pythagorean theorem tells us that for a leaf node of size $n \times n$, the maximum number of elements we need to consider is $\sqrt{2}n$ (the hypotenuse of the node area), assuming $n \geq 0$.

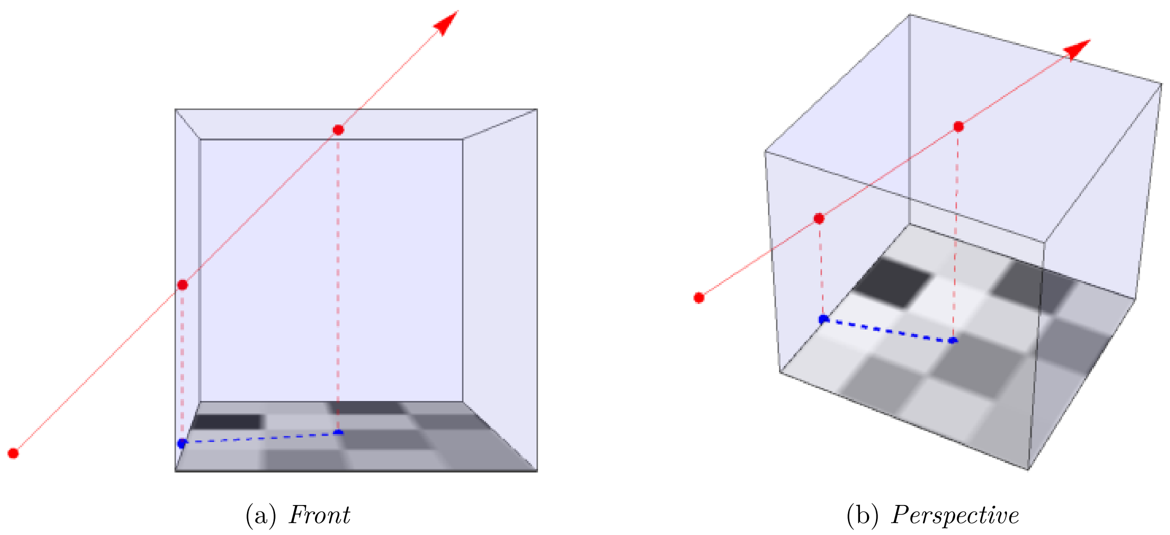
Since the entry and exit point of every AABB is known, we can construct the line L passing through the AABBs as:

$$p_{entry} = ray_{origin} + (ray_{direction} \times t_{min})$$

$$p_{exit} = ray_{origin} + (ray_{direction} \times t_{max})$$

$$L = \{p_{entry}, p_{exit}\}$$

If this line is projected onto the 2-dimensional leaf area in the height map, this new line intersect the (and only the) elements we need to consider for ray-terrian-intersection (as seen in Figure 3.1).



(a) *Front* (b) *Perspective*

Figure 3.1: *A ray (red line) has intersected a node leaf of* 4×4 *elements and the segment of the ray inside the leaf has been projected onto the height map plane (blue line). Only 2 of the 16 elements need to be considered.*

Using the algorithm proposed by Bresenham [1] we can efficiently find the height map elements that lie along the projected line. Also, if we choose to begin considering elements starting from p_{entry}

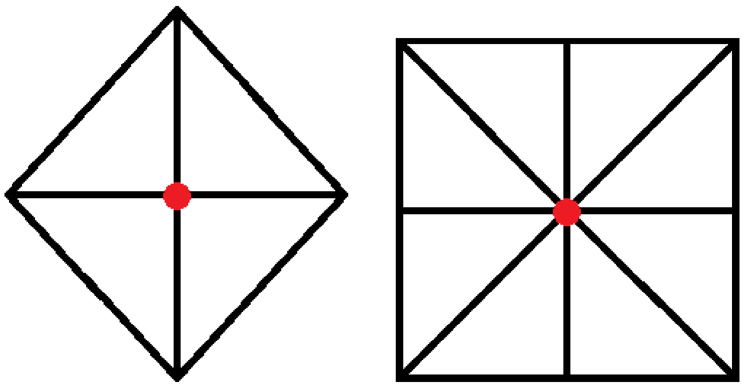
and ending at p_{exit}, we can be certain that the first intersected height map element is the correct one and stop considering additional elements or quadtree nodes.

3.5 Height map element intersection

For every intersected height map element, the triangles corresponding to that element must to be tested for intersection with the ray and sorted with respect to the distance of the intersection point and the ray origin. The triangle with the shortest distance is the actual intersected triangle, as illustrated by the following pseudo code (including line walking):

```
for all $\{x, y\}$ between $\{p_{entry}, p_{exit}\}$ do
  $near_{triangle} \leftarrow none$
  $near_{distance} \leftarrow none$
  for all $triangles$ at $\{x, y\}$ do
    $\{intersected, t\} = RayVsTriangle(ray, triangle)$
    if $intersected =$ true then
      if $t < near_{distance}$ then
        $near_{triangle} \leftarrow triangle$
        $near_{distance} \leftarrow t$
      end if
    end if
  end for
  if $near_{triangle} \neq none$ then
    return $near_{distance}$
  end if
end for
```

What triangles correspond to a height map element naturally depends on how triangles are created from the source matrix. For example: in the suggestion given in 2.1: for any given point $\{x, y\}$ in the height map there are two discrete cases for the triangles pertaining to that point:



(a) *Case A: 4 triangles* (b) *Case B: 8 triangles*

Figure 3.2: *Possible triangles of any point $\{x, y\}$ in the height map using the method described in 2.1*

Technically, there are edge cases for vertices on the first and last row or column in the matrix; for instance if the top-left element in the height map would yield the triangles in Case A, only the triangle in the fourth quadrant would be valid (since the others would contain vertices that are out of bounds).

Once the triangles for a height map element have been identified, the ray is tested for intersection against them. If one or more triangles intersect the ray the intersection point closest to the ray origin is immediately returned as the point where the ray intersects the terrain (it may also be interesting to return the surface normal of the intersected triangle if a return direction vector is useful).

If no triangle is intersected, the algorithm continues walking the line as in Section 3.4 until one is found. If none is found, the ray does not intersect the terrain geometry.

3.6 A note on optimization for Z-only rays

For rays with a direction strictly on the Z-axis, such as $\{0, 0, \pm z\}$, many of the steps described previously can be skipped. In this case, a coordinate transform from the ray origin into the height map, followed by intersection tests for the triangles mapping to that single height map element is sufficient. This can be a relevant optimization for repeatedly updating the position of objects placed on the terrain.

4 Results

Three different methods of finding the intersection point of a ray and terrain geometry have been compared in a worst-case scenario, where a ray intersect all quad leafs along the hypotenuse of the terrain geometry (the maximum number of triangles need to be investigated). The methods are implemented in C++ and operate on the same backing data.

Table 4.1 shows a comparison of speedup in execution time between the different methods, normalized against a reference method (the ”No pruning” method):

- *No pruning*: testing all triangles contained in the geometry mesh without pruning. Reference method.
- *Quadtree only*: using a quadtree to search the geometry mesh, but testing all triangles contained in leaf nodes (without line walking).
- *This method*: the method proposed in this paper.

Height map size	**Leaf size**	**No pruning**	**Quadtree only**	**This method**
1024×1024	32×32	1.0	11.9	743.2
2048×2048	32×32	1.0	21.8	1317.9
4096×4096	32×32	1.0	43.6	2298.5
8192×8192	32×32	1.0	83.0	3966.0

Table 4.1: Comparison of speedup in worst-case intersection test calculations, compared to the reference method in the third column.

5 Conclusion

To efficiently calculate the intersection point of a ray through a large terrain geometry mesh, the key to achieving good performance is to reduce the number of triangles that need to be tested.

The traditional approach of using a quadtree to search a height map for possible intersections greatly reduces the number of calculations required. However, the number of calculations can be further reduced by projecting the line intersecting each quadtree leaf node onto the height map plane (as shown in Section 3.4) due to how the only elements that need to be considered lie along that line which can be rasterized efficiently using Bresenham's line algorithm. Since the maximum length of a line through a rectangle is the hypotenuse of that rectangle, this effectively reduces the complexity of the intersection algorithm for an $n \times n$ leaf node from $O(n^2)$ to approximately $O(\sqrt{2}n)$.

As seen in Table 4.1, the method proposed in this paper is favorable to the traditional approaches in speed. It does not require actual world-space triangles to be stored in memory, and relies solely on the source height map matrix to operate. This is likely an acceptable compromise, since this matrix can be used by a vertex shader to render the geometry.

The main drawback of the method is the maintenance of the quadtree data structure; when the height map changes the quadtree needs to be recomputed. With knowledge of the affected region however, the update should be able to be done efficiently (see Future work) without having to recompute the entire tree.

5.1 Future work

There are still improvements that can be done to the algorithm:

- When walking the projected line in the height map using Bresenham's line algorithm (as discussed in Section 3.4) more triangles than necessary are being tested. By observing the δx and δy when moving along the line, one can make assumptions about what triangles were tested at the previous coordinate which may overlap with the triangles at the current coordinate. Implementing this optimization could reduce the number of triangles required for consideration by up to 25% in the average case at the cost of a few conditional statements.

- If the height map is modified after the quadtree has been computed it should be possible to modify only the branches of the quadtree that were affected and trickle the change up through parent branches, instead of recomputing the tree completely.

References

[1] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal* **4**.1 (1965), 25–30.

[2] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-triangle Intersection. *J. Graph. Tools* **2**.1 (Oct. 1997), 21–28. ISSN: 1086-7651. DOI: `10.1080/10867651.1997.10487468`. URL: `http://dx.doi.org/10.1080/10867651.1997.10487468`.

[3] B. Smits. Efficiency Issues for Ray Tracing. *Journal of Graphics Tools* **3**.2 (1998), 1–14. DOI: `10.1080/10867651.1998.10487488`. eprint: `http://dx.doi.org/10.1080/10867651.1998.10487488`. URL: `http://dx.doi.org/10.1080/10867651.1998.10487488`.

[4] A. Williams et al. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools* **10**.1 (2005), 49–54. DOI: `10.1080/2151237X.2005.10129188`. eprint: `http://dx.doi.org/10.1080/2151237X.2005.10129188`. URL: `http://dx.doi.org/10.1080/2151237X.2005.10129188`.