Automated generation of waypoints
for pathfinding in a static environment

Magnus Nyqvist
Fredrik Olsson
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Magnus Nyqvist
E-mail: makp16@student.bth.se

Fredrik Olsson
E-mail: frol16@student.bth.se

University advisor:
Senior Lecturer Dr. Parshant Goswami
Department of DIDA

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

Video game characters must almost always be able to travel from point A to point B and this task can be solved in various ways. There exist grid maps, waypoints, mesh navigation and hierarchical techniques to solve this problem. On randomly generated terrain we make use of automatically generated waypoints to solve pathfinding queries. The waypoints are connected by edges to create a waypoint graph and the graph can be used in real time pathfinding for a single agent in a static environment. This is done by finding the vertices of the blocked triangles from the terrain and place a waypoint on each. We make use of the GPU to create the waypoint graph. The waypoints are connected by utilizing a serialized GPU quad tree to find the relevant blocked geometry to do a line-triangle intersection test. If a line between two waypoints do not intersect any blocked geometry the connection is valid and stored. We found out that it is possible to generate a waypoint graph during the startup of the game with acceptable time results, make use of such a graph for real time pathfinding and that players perceive the paths generated by the algorithm as realistic and responsive. Our conclusion is that our solution is well suited for a deterministic environment with agents of the same size.

Keywords: Pathfinding - Waypoint generation - Terrain generation - User perception
Acknowledgments

We would like to give our greatest gratitude towards everyone who participated in our user experiment. A big thank you to Mats-Ola Landbris, Hans Tap and Martin Lillqvist who helped us distribute our experiment to the participants. We would also like to thank our supervisor, Parshant Goswami, for all the valuable feedback during the creation process. Lastly, we would like to thank Wilhelmina Leopoldson Fallkvist for proofreading our thesis.
## Contents

Abstract                                           i
Acknowledgments                                    iii

1 Introduction                                      1
   1.1 Background                                    1
   1.2 Research questions and objectives            2
   1.3 Thesis structure                              2

2 Related Work                                      3

3 Method                                           5
   3.1 Automatically generated waypoints             5
      3.1.1 Input and output                           5
   3.2 Research method (A)                           6
      3.2.1 Terrain Generation                        6
      3.2.2 Creation                                  6
      3.2.3 Cleaning                                  8
   3.3 Research method (B)                           9
      3.3.1 Creating the waypoint graph               9
      3.3.2 Pathfinding Algorithm                     13
      3.3.3 User study                                14

4 Results                                           17
   4.1 Research question (A)                         18
      4.1.1 Waypoint Generation results               18
      4.1.2 Results of removing unnecessary waypoints 19
   4.2 Research question (B)                         19
      4.2.1 Connection Reduction Results              19
      4.2.2 User Study                                20

5 Analysis and Discussion                          23
   5.1 Map generation                                23
   5.2 Waypoint generation                           23
   5.3 User perception                               24
Chapter 1

Introduction

1.1 Background

In almost every video game, a character needs to travel from point A to point B and this problem needs to be solved if the player is not in total control of the character. Pathfinding in video games has to be done in real time and the speed of the pathfinding algorithm is limited to the computer hardware[1] and the algorithm optimization. Depending of the complexity of the video game, there exist different solutions to the pathfinding problem. The most common approach is the use of grids, mesh navigation, waypoints or some hierarchical techniques such as quad trees[2].

In a grid map, the game map is divided into small squares, hexagons or triangles and is referred to as nodes. A node can be either blocked or unblocked. If a node is unblocked it is possible to walk on top of the area it represents and if the node is blocked this is not possible. Each node have an edge which describes how to go to the next node. In a square grid every node has up to eight edges; up, up-right, right, down-right, down, down-left, left and up-left. The edges are defined by the neighbouring nodes to the current node. A variant of the A* algorithm is the most common grid traversal algorithm[12]. The A* algorithm is evaluating the current node and picks the next node by comparing the edges to each other. The edge which leads to a node with the smallest total cost will become the next current node to pick. This step is repeated until the goal node has been found. The total cost is determined by the heuristic used in the algorithm but is usually calculating by the total distance traveled, distance to the goal node and the distance to the next node.

Mesh navigation (NavMesh) has the same principle as a grid map[3] but instead of using two dimensional squares it makes use of three- or two dimensional polygons that do not necessarily needs to be of the same size. These polygons provides a map over the unblocked areas of the terrain and is traversed in almost the same manner as a grid map.

A hierarchical map represents the terrain in different level of detail. On what level the path should be calculated on can depend on various variables. The basic principle is that the path is calculated roughly on a high level and then during the path, the algorithm is invoked again on a more detailed level[3]. This divides the path into different areas of the terrain and in pathfinding algorithms, shorter paths are much faster to calculate than longer paths.

Waypoints is another approach to this problem. Waypoints represent important locations on the terrain, such as passages or edges of blocked geometry. A line drawn in between two waypoints is called a connection or an edge. A waypoint graph that
is generated based on visibility is called a visibility graph[16]. A visibility graph is in practice an inefficient approach, due to the high memory consumption and increased computational cost of the pathfinding process[14]. Connection sparsification is therefore desired to increase the usability of the waypoint graphs in larger environments.

To be able to use pathfinding on randomly generated terrain the terrain needs to be represented by a graph of some kind. The techniques to generate this graph vary depending on the complexity of the terrain and the goal of the application. In this thesis we want to find out if it is possible to automatically generate a waypoint graph from randomly generated polygon soups for a single agent in a static environment. Also we want to evaluate users perception of pathfinding in such a waypoint graph.

1.2 Research questions and objectives

Our aim is to automatically generate waypoints on random map layouts. It is our intention to make the waypoints usable for pathfinding in video games such as the real-time strategy (RTS) games genre. The waypoints will be created in a two-and-a-half-dimensional environment and get traversed by a variant of A*. For this thesis there are two research questions to be answered.

(A) Is it possible to automatically generate waypoints to be used on random maps?

(B) Is it possible to traverse automatically generated waypoints in a way that users perceive the result as realistic and responsive?

To answer these thesis questions, the following objectives have to be realised.

(i) Generate random map layouts to create the game environment.

(ii) Generate waypoints from the game environment. The waypoints will be based on the maps blockage.

(iii) Measure pathfinding responsiveness and path quality based on user perceptions.

1.3 Thesis structure

The organization of the remainder of the thesis is as follows. In chapter 2 we introduce the related work to automatically generate waypoints, where we show other approaches to be able to use pathfinding on randomly generated terrain. After that in chapter 3 we present our research method on how to create automatically generated waypoints, to answer research question (A). We also present how we will use a user experiment to receive good feedback on the quality of the paths generated, to answer research question (B). Next in chapter 4 the results of our implementation and user study is presented. We present the results of our research questions and present the gathered data. Analysis and discussion are thereafter found in chapter 5, where we analyze our objectives and discuss the data presented in chapter 4. Lastly, in chapter 6 we talk about our conclusions of our work and possible future work.
Chapter 2

Related Work

Map representations can be automatically generated with a number of different methods. Lozano-Perez and Wesley’s *visibility graph*\[16\] defines waypoints at the vertices of obstacles. The waypoints are connected by straight lines, also called edges, based on visibility. Visibility is defined as every pair of waypoints that has an intersection free edge in between them. Pathfinding queries can be answered based on the connectivity information stored in the graph and the result will be a collision free path for a point character.

Another similar approach is the *balanced waypoint graph*\[17\] suggested by Weiping et al.. Waypoint candidates are defined at each vertex of an obstacle, and then displaced outwards with the distance $r$. The distance $r$ is used for further comparison of distances in the explanation of the algorithm. The waypoint candidates are then evaluated, to check their validity. A waypoint candidate is considered to be valid if there are no obstacles intersecting a circle, with radius $r$, around the waypoint. All invalid waypoint candidates are removed and the remaining candidates are clustered. Clustering is done by dividing the region where the generation takes place into a $2r \times 2r$ grid, and waypoint candidates that reside in the same grid square are clustered. Clustered waypoint candidates result in one waypoint at the center, or close to the center, of the grid square. Edges are then generated based on visibility and character size. Waypoint pairs are connected by edges if the corridor in between them is collision free. The last step to achieve the balanced waypoint graph is to build a spares waypoint graph from the current graph. The sparsification is done by putting a square around every waypoint in the graph, and then determining what edges are important to the pathfinding. This is done by grouping the edges of a waypoint to the square edge they belong to and then evaluating the paths they can generate. An edge can be discarded if there is another path to the same destination, through multiple edges, and the direction change as well as the traversal distance is not significantly longer. The graph generation is complete when the sparsification is done.

Wardhana, Johan and Seah suggests a method to generate an *enhanced waypoint graph*\[14\]. The enhanced waypoint graph is similar to the balanced waypoint graph, but it can handle characters of many different sizes. It is also a solution that is made for volumetric pathfinding. The algorithm uses voxels instead of vertices to determine waypoint candidates. Voxelization is memory consuming and the enhanced waypoint graph is therefore generated with environmental subdivision when large virtual worlds are sent as input data. The environment is divided into a number of regions that generate their own waypoint graphs. The regional waypoint graphs are later connected by special border waypoints, to generate the enhanced waypoint graph.
Waypoint graphs are not the only graphs used for pathfinding. Other approaches have been taken to solve the problem of creating graphs for path planning, such as square grids and mesh navigation\cite{2}. Square grids can have a maximum of eight neighbours and there are numerous proposals of algorithms that solve pathfinding problems on square grids. One of the proposed algorithms is the jump point search (JPS) algorithm\cite{6}. Harabor and Grastien developed the algorithm to reduce the unnecessary evaluation of equivalent states during pathfinding and JPS is around ten times faster than A*, when evaluated with the standard dataset of pathfinding maps proposed by Sturtevant\cite{13}. Both square grids and mesh navigation are commonly used by computer games, and in addition to that square grids are also a popular graph representation in robotics. Mesh navigation graphs utilize a mesh to represent the walkable areas of a map\cite{2}. The mesh can be represented by polygons and it looks very similar to the visibility graph. The difference is that a visibility graph is more complex than a mesh graph, when it comes to practical terms. Polygonal navigation meshes can greatly accelerate pathfinding compared to regular grids\cite{15}, even though they are simpler than visibility graphs. The navigation mesh can represent geometry with fewer cells than a regular grid, and it is beneficial to have a lower amount of cells during pathfinding.
3.1 Automatically generated waypoints

The core of this thesis is to input a terrain to our algorithm and as an output, get automatically generated waypoints. The steps to accomplish this can be seen in Figure 3.1. We need to define blocked triangles from the terrain, place a waypoint on every vertex of every blocked triangle, put blocked geometry and waypoints into quad trees, clean unnecessary waypoints and then connect the remaining waypoints to each other. The graphical representation is viewed in three dimensions but behind the scenes, everything is defined in two dimensions, which makes up the two-and-a-half-dimensional environment. The unit can only walk in the X and Z axis and the terrain will offset the unit based on the current X and Z coordinates. This is due to the simplicity of pathfinding in a two dimensional environment. The unit dispatches a ray downwards from its current location to find an intersection point in the terrain, and the Y value is extracted from that intersection point.

3.1.1 Input and output

Our waypoint generation algorithm requires one input data and that is a polygon soup representation of a two-dimensional virtual world. This limits the algorithm to non-volumetric virtual worlds.

The output is a weighted waypoint graph consisting of waypoint nodes and edges connecting them. Each waypoint represents important environmental locations such as corners and entrances to passages. Edges hold one variable and that is their
3.2 Research method (A)

To be able to answer research question (A) the existence of waypoints is needed. They should be placed based on the randomly generated terrain and there should be waypoints left after the cleaning is done. To prove the existence of waypoints, we will input different types of terrain to our algorithm and sample the waypoint count before and after the cleaning for each iteration for a total of 20 iterations. The waypoint generation and waypoint cleaning steps will realise objective (ii).

3.2.1 Terrain Generation

A diamond-square algorithm (DSA)[7] is utilized to create various test environments, also referred to as maps. The output of the DSA is a list of height values, which are later converted into a polygon soup and input into the waypoint generation algorithm. A post process Gaussian blur[11] was added to the DSA to reduce extreme height differences in the height map. The maps are limited to a size of 1025x1025 units. This will realise objective (i).

3.2.2 Creation

The idea of the creation of waypoints is inspired by the work of Weiping et al.[17]. The input of our algorithm is a randomly generated polygon soup, which is a list of triangles. The first step is to define blocked triangles in the triangle list. If the triangle has a normal which is too steep, this triangle is defined as blocked. The normal is defined as too steep if the angle from the dot product between the normal
3.2. Research method (A)

(a) Pre-smoothing blockage
(b) Post-smoothing blockage

Figure 3.3: Smoothing algorithm explained from the initial blockage (a) to the final blockage (b). The algorithm checks every unblocked triangle and blocks any triangle that passes the test of being unnecessary. The red triangles are triangles that the algorithm blocked. The pink circles in figure (b) are the points that the algorithm uses to determine if the green triangle should be blocked or not. A triangle is deemed unnecessary and becomes blocked if three or more points are contained by blocked triangles. The gray geometry is defined as blocked.

and the up vector is below 45 degrees. Also if a vertex belonging to a triangle has a Y-value below sea level, this triangle is also defined as blocked. The sea level is a simple float value which determines where the water model and texture should start and we use a value of -10. To lower the amount of waypoints and also help the cleaning algorithm we also block irregular shapes (Smoothing) based out of the true blocked triangles. The smoothing provides a method to reduce the amount of waypoints without loss of precision when executing pathfinding. Irregular shapes generate lots of waypoints that the cleaning algorithm cannot remove. Figure 3.3 displays a case where an irregular shape would create unnecessary waypoints, and how the smoothing algorithm modifies the triangle blockage to avoid it. The pathfinding is not improved by keeping the waypoints, but the cleaning algorithm will not remove them because they are considered to be geometry gaps and the pathfinding could lose precision if they were removed. A smoothing algorithm is therefore executed prior to the waypoint cleaning.

The smoothing algorithm block unblocked triangles that does not improve the pathfinding and this is determined by checking four points around the triangle. The points are found by calculating the center point of the current unblocked triangle and then offsetting it by one step up, right, down and left. A step in the coordinate system is defined as 1.0. If three or more of the points are blocked then the current triangle also becomes blocked.

When all of the blocked triangles are defined, the terrain triangle list is placed in a quad tree to be used during the connection step and traversal. For every blocked triangle in the list, a waypoint is added on each vertex to a map data structure and the original creation position will be used as the key. If there already exist a
waypoint that has the same key, no new waypoint will be added to the map. All waypoints are now placed on blocked geometry, see Figure 3.2a, which in this current state, will not generate any connections to other waypoints. The solution is to offset the waypoints half a unit in the direction from the center of the triangle towards the waypoint position as seen in Figure 3.2b. The offset is half a unit because the agent has the size of one unit, so half the agent size is half a unit. The waypoint offset is done after the cleaning of the unnecessary waypoint.

3.2.3 Cleaning

Waypoints are created at every blocked vertex, and there are many unnecessary waypoints after the initial creation. It is therefore necessary to remove redundant waypoints. Cleaning is done in the five following steps: Initial blocked cleanup, remove x-axis aligned waypoints, remove y-axis aligned waypoints, remove diagonal aligned waypoints and waypoint clustering.

Initial blocked cleanup

An initial removal of waypoints is required to reduce the density of the waypoints for further cleanup. Waypoints that are created on completely blocked geometry is removed in this step. A waypoint is marked for removal if the waypoint has blocked geometry one step up, right, down and left in the coordinate system. The blue waypoints in Figure 3.4 are removed by this.

Remove x-, y- and diagonal aligned waypoints

The remaining waypoints are now positioned along the edges of the blocked geometry. Only the two key waypoints, at the beginning and the end of the edge, are required for pathfinding. Waypoints are marked for removal during this stage if they are continuously aligned and in between a start and an end waypoint.
All the alignment cleanup steps take geometry gaps into consideration. A passage in between two blocked areas would lose its waypoint without gap checks. The algorithm executes blocked triangle intersection tests at every step forward to make sure that it is following an edge. Red waypoints in Figure 3.4 are removed by this step.

**Waypoint clustering**

The last part of the cleanup sequence is to remove waypoints that are located too close to each other. This is done by checking for waypoints in a 2x2 square around the current waypoint. Any waypoint contained by the square is marked for removal. The 2x2 square is used to get all waypoints around the current waypoint (one unit in every axis) to see if there should be any clustering.

### 3.3 Research method (B)

For research question (B) to be answered we need to be able to use the waypoints for pathfinding. To make this possible we must first and foremost create a waypoint graph of our waypoints, by connecting the existing waypoints to each other. We must also create an algorithm to traverse the waypoint graph in an optimal way, so that the paths are calculated quickly, but also generate a qualitative result. Lastly, the waypoint graph must be tested on actual users to get the relevant user perception data. The later mentioned data is needed to prove that our waypoint graph generates responsive and qualitative paths. Generation and testing of the waypoint graph will realise objective (iii).

#### 3.3.1 Creating the waypoint graph

The rule for a waypoint to be connected to another waypoint is that the connection needs to be in line of sight, i.e there exist a line between two waypoints which is not intersected by any blocked geometry. To connect all the waypoints to each other have a Big O notation of $O(n^2)$, where $n$ is the number of waypoints.

In our solution, where every waypoint can potentially be connected to any other waypoint in the scene, we found out that it was better to use the GPU with the use of threads instead of the CPU to find the connections simultaneously. This is still too expensive by itself to be done using pure line-triangle intersection tests. To make it possible we make use of a serialized GPU quad tree to find relevant triangles to do the line-triangle intersection test while finding the connections. The result from the GPU is a waypoint connection map. This map is used to connect the waypoints to each other.

To lower the memory usage, there exist an approximation connection rule. When adding a new connection to a waypoint, if there already exist a connection which has almost the same direction as the one we try to add, the connection which has the lowest cost (shortest length) is stored. This is done because the assumption can be made that if two connections with almost the same direction exist, the connected waypoint with a lower cost will be connected to the waypoint which has the higher cost. This is one method of sparsification.
Chapter 3. Method

Figure 3.5: How the waypoints get connected to every other waypoint on the terrain. The gray geometry is defined as blocked.

In Figure 3.5 the waypoint A, B and C shows this connection rule. Waypoint A is not directly connected towards waypoint C but have an indirect connection via waypoint B. This happens because the connection A to C has the same direction as A to B but the A to B connection cost is lower and therefore this connection is chosen. The connection approximation is needed to lower the number of connections from each waypoint. It is better to make the assumption that the neighbour waypoint have a connection towards the waypoint further away to not make the waypoint graph to dense. A waypoint graph with a high density is memory consuming and also expensive to traverse[14].

Serialized GPU Quad Tree

The problem of using a quad tree on the GPU is that the quad tree needs to be serialized. It is easy to use a quad tree on the CPU where the tree can be scattered in the memory and store pointers to the geometry which is contained in the nodes. Our shaders are programmed using High Level Shader Language (HLSL)[9] and when using buffers on the GPU, everything needs to be serialized. On the CPU, a tree node is allocated somewhere in the memory and the triangles inside that node is not allocated directly after that node. This node only has a pointer to where in the memory the triangles are allocated. To serialize the quad tree we copy the node to a buffer and then copy the containing triangles to the memory address directly after the node memory ends. This means that all the nodes will have different sizes and a structured buffer[10] can therefore not be used. Instead we use a Byte Address Buffer[8] which is accessed manually using byte address values. Every node stores the byte address start, size and it’s children byte address start. The source code of the serialization can be seen in Appendix A.1.

Data structures used on the GPU

To find the waypoint connections via the GPU we need the blocked geometry, the waypoints and the serialized quad tree. The blocked geometry and the waypoints
3.3. Research method (B)

are uploaded to the GPU through structured buffers and the quad tree to a byte address buffer. The byte address buffer is now containing the tree nodes and indices to the blocked geometry buffer. A tree node data structure defines a tree node. The tree node stores the byte address to the node, the minimum and maximum position of the axis aligned bounding box (AABB), number of children of the node, the byte address to the children and the number of triangles the AABB contains. It is good to point out that in our solution, the leaf nodes is the only nodes that can have triangles inside of them. This means, if the node do not have children, it is a leaf and might contain triangles.

While traversing the quad tree, each traversed node are put on a stack as a stack element. Each stack element is used to store current nodes while traversing. The variables of a stack element is the node byte address which is an index to the byte address buffer. It also stores what children to target next which describes what children has already been traversed. For example if this variable has the value of 0, no children has been traversed. If the Target children has the value of 2, children number zero and number one has been traversed and the next children is children number two. When all children have been traversed, this node is popped from the stack.

To extract a tree node from the byte address buffer we need to manually offset the memory to get all the structure data. Every time an element is extracted the address needs to be moved. The source code can be seen in Appendix A.3.

The GPU waypoint structure is the return value to the CPU. It defines for every waypoint a possible connection to all other waypoints, with a maximum of 256 connections. It contains the index of the waypoint, number of connections, the waypoint position and an array of all the connections for this waypoint. The source code of the data structures used can be seen in Appendix A.2.

Connection Definition Traversal Algorithm

Using the GPU is a good solution for heavy calculations which can be done in parallel and we batch the dispatches of threads to not over consume the GPU. The work is batched so the GPU do not suspend itself due to long execution time. There is a way to disable the timeout of the GPU[4], but we decided not to do this to maintain our real-time requirement.

In each iteration of batched work an offset buffer is bound to the GPU which offset the thread ID used to determine what waypoints to target and dispatches 512 thread groups. Every thread group will calculate the connection for one waypoint i.e. each iteration will find the connections for 512 waypoints. The source code for the batched work can be found in Appendix A.4.

Inside each dispatched thread group there exists 1024 threads. Each thread in each group will have a specific task to do, to maximize performance. In the shader, every thread in the thread group will try to connect the target waypoint against different waypoints.

To find what waypoint index the thread group should target (wpt), the thread group ID (gid) and the previous offset (of) is added together wpt = gid + of. The group thread ID (gtid) is used to determine what waypoint in the waypoint structured buffer this thread should start finding connections towards.
To extract how many waypoints each thread should test against \((nrwp)\), the total number of waypoints \((towp)\) is divided with the group size \((1024)\) with a minimum of one waypoint \(nrwp = towp/1024\). The waypoint each thread should start testing against \((wps)\) is calculated based on the current waypoint target due to that no check is needed against a waypoint that is "behind" the current waypoint target in the structured buffer. If a connection exists the connected waypoint has a connection towards the other waypoint as well. The formula used to find the start waypoint is \(wps = nrwp \times gtid + wpt + 1\). The last waypoint each thread should test against \((wpe)\) is calculated by adding \(wps\) with \(nrwp\). \(wpe = wps + nrwp\) with a max index of \(towp\).

For every waypoint this thread should test against, a line is defined which always has its origin at the target waypoint (source) and ends at the waypoint the source tries to connect to (destination). If the length of the line is above 128 units there exist no connection, even though this line may not intersect any blocked geometry. This limitation exist to limit the number of connections of each waypoint. Source code can bee seen in Appendix A.5

The root node of the quad tree is pushed on the stack. While the stack size is above Zero and no intersection with geometry has been found, we check if there are potential children of the top node in the stack. If the top node do not have any children to traverse it is popped from the stack and otherwise, if the node has children, the line and the children node AABB is tested for an intersection. If the line and child node intersects with each other and this child has triangles the line is tested for intersections against every triangle inside the node until an intersection is found. We are only interested if there exist an intersection between the line and blocked geometry. There is no need to know exactly where the intersection takes place, how close the intersection is nor how many polygons the line intersects. If the child node do not have any triangles it is pushed to the stack. When every potential polygon has been tested against and still no intersection has been found, a connection between the two waypoints exist and is saved to the target waypoint. The source code can bee seen in Appendix A.6 and A.7

**GPU Output**

Back on the CPU the waypoint buffer is mapped[5]. For every waypoints’ connections, defined from the GPU, the two waypoints in the connection are connected to each other. The waypoint which has defined the connection is connected first. It is not for certain that this connection is a true connection because we do not want a pure waypoint visibility graph[16]. For a connection to be added it has to pass a direction test.

The cost to travel via this connection is defined by the length of the line between these waypoints. For every existing connection, for the target waypoint, the new connection is tested. The test is a dot product between the existing connection and the new connection. If there exist a connection which has almost the same angle, a dot product above 0.9, only the one connection which has the lowest cost is stored. This is because we can assume that if two connections has the same angle, the one with the lower costs’ destination will also have a connection to the destination of the connection with the higher cost. If the connection is added we "force connect" the
3.3. Research method (B)

destination with the source which means that there will always be a way back from the destination to the source. When all waypoints are connected, every waypoint that do not have a connection gets removed.

3.3.2 Pathfinding Algorithm

The pathfinding algorithm is A*\textsuperscript{2} based, but the first path is calculated instead of the shortest. The input variables to the algorithm is the source position, destination position and a quad tree which contains the waypoints and the blocked geometry. Before the algorithm begins a check to see if the destination is in line of sight from the source position. If this is the case, the destination position is returned to the requester due to the existence of a direct path. If not, the algorithm takes place.

An unordered map data structure is used to store all visited waypoints and a map data structure is used as the open list. The open list contains the nodes that is the most possible nodes to traverse next to come closer to the goal. The visited unordered map represents the nodes already visited and in our algorithm, a node can only be visited once. It is important to point out that the elements in a map data structure is sorted based on the key value and the elements in the unordered map data structure is not. The key to the open list is a node which is sorted by the total cost to travel to this node which gives the result that the first node in the open list will always be the cheapest and most desirable. A node in this case is a structure which contains the waypoint it represents, the waypoint parent and the total cost to go to this node.

The first step is to get the closest waypoint to the source position. While no start waypoint has been found, all waypoint which is inside a radius (which expands every time no start point has been found) from the source position is tested for direct line of sight. The closest waypoint to the source which meets the requirements will be set as the start waypoint.

While the open list contains elements, the current node is set as the first node in the open list and is erased from the map. If the current node has a direct line of sight to the destination position there exist a path. Else, all the connections from the current node is checked against the visited unordered map data structure and if they are not set as visited they are inserted as new nodes inside the open list and marked as visited. The algorithm then starts over and extracts the first node in the open list. When a path to the destination is found, the path rebuild itself by going backwards from the current node via the parents until there no longer exist any parent. The path generated is not the shortest path due to this being unachievable in real time with our waypoint count. When calculating the path with the best path in mind, every node has to be visited from every angle to see which node has the shortest distance. Instead we make use of the first path found.

Path Enhancing

The paths generated by waypoint do not generate the best visual results due to the straight lines between each waypoints. The units walk straight to the next waypoint and then do a sharp turn towards the next. There will also sometimes occur a "back-and-forth" behaviour if two waypoints are close to each other and this behaviour is
not desired in turns of visuals. To remove this behaviour when the unit is using the
path during run-time, a ray towards the next position is dispatched in each update to
see if the unit can skip the current destination waypoint. This creates the behaviour
of a smooth walk instead of a sharp, straight line, unnatural behaviour.

3.3.3 User study

A user study is needed to evaluate the waypoint graph from an end user perspective.
The important part is to evaluate if the generated paths are experienced as natural
and responsive. The game environment we present the test subject with during the
experiment can be seen in Figure 3.6. The goal for the subject is to move a character
around in the environment and collect yellow boxes. The controls are similar to
those of a real time strategy (RTS) game, and they are presented to the user in a
separate file. It is possible to zoom, move and rotate the camera, as well as moving
the game character Jonny through right-clicking positions in the game environment.
The game generates new terrain and starts over when all boxes are collected.

It is important to note that the test subject do not possess the knowledge of
what they are testing. They are asked to play the game and then fill a survey. The
subject group of choice is people with some or more gaming experience. We made
the choice of group because the answers from a group with some knowledge of the
area will give more relevant answers. A person which has no experience what so ever
of video games might not think in terms of movement while playing and will not
analyze what is happening on the screen. They will probably struggle with the user
input and will only focus on that part of the game.

The test is sent out to the subjects by email and is not supervised by anyone
but themselves and they only posses a set of instructions. The instructions used can
be found in Appendix B.1. The survey consist of six sections which target different
parts of the game. The only section we really care about is the pathfinding section,
in which we ask the user what they think of the pathfinding. Under this section
we ask the user how they perceive the movement, if the controllable cube walked as
they intended and if the cube took paths that they did not feel were the right ones.
Each question can be answered by select an answer on a linear scale of 1 – 5 and the
subject can also leave a comment about the pathfinding. The survey can be found
in appendix B.2.
Figure 3.6: The environment used in the user experiment. The small red box in the center of the figure is the player controllable unit and it is named Jonny. The gray area and the water is unwalkable surface. The yellow boxes are collectables. The goal of the test from the user perspective is to collect all yellow boxes.
Chapter 4

Results

(a) Input data: Polygon soup (generated terrain)  (b) Define blocked (red) and unblocked (green) geometry
(c) Create waypoints on every vertex of blocked geometry (black boxes)

(d) Remove unnecessary waypoints  (e) Offset waypoints towards unblocked geometry

Figure 4.1: Automatically generated waypoints (f) based on the input terrain (a). From the input data (a), the algorithm defines blocked geometry based on the triangle angle (b). In every vertex of the blocked geometry a waypoint is placed (c). The unnecessary waypoints gets removed from the blocked geometry (d) and the remaining waypoints are offset from the blocked triangle center to unblocked geometry (e). The waypoint dispatches a ray towards all other waypoints and if the ray do not intersect any blocked geometry a connection exists (f).
<table>
<thead>
<tr>
<th>#</th>
<th>Pre-cleaning</th>
<th>Post-cleaning</th>
<th>Remaining (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>381800</td>
<td>26081</td>
<td>6.83</td>
</tr>
<tr>
<td>2</td>
<td>383413</td>
<td>25579</td>
<td>6.67</td>
</tr>
<tr>
<td>3</td>
<td>513623</td>
<td>16680</td>
<td>3.25</td>
</tr>
<tr>
<td>4</td>
<td>825430</td>
<td>9275</td>
<td>1.12</td>
</tr>
<tr>
<td>5</td>
<td>438353</td>
<td>25089</td>
<td>5.72</td>
</tr>
<tr>
<td>6</td>
<td>601700</td>
<td>10829</td>
<td>1.80</td>
</tr>
<tr>
<td>7</td>
<td>296019</td>
<td>22209</td>
<td>7.50</td>
</tr>
<tr>
<td>8</td>
<td>487508</td>
<td>15371</td>
<td>3.15</td>
</tr>
<tr>
<td>9</td>
<td>701996</td>
<td>9883</td>
<td>1.41</td>
</tr>
<tr>
<td>10</td>
<td>386112</td>
<td>25681</td>
<td>6.65</td>
</tr>
<tr>
<td>11</td>
<td>669985</td>
<td>8434</td>
<td>1.26</td>
</tr>
<tr>
<td>12</td>
<td>533584</td>
<td>21562</td>
<td>4.04</td>
</tr>
<tr>
<td>13</td>
<td>666100</td>
<td>12460</td>
<td>1.87</td>
</tr>
<tr>
<td>14</td>
<td>469535</td>
<td>26203</td>
<td>5.58</td>
</tr>
<tr>
<td>15</td>
<td>568589</td>
<td>16807</td>
<td>2.96</td>
</tr>
<tr>
<td>16</td>
<td>723765</td>
<td>12952</td>
<td>1.79</td>
</tr>
<tr>
<td>17</td>
<td>517824</td>
<td>28804</td>
<td>5.56</td>
</tr>
<tr>
<td>18</td>
<td>603208</td>
<td>17547</td>
<td>2.91</td>
</tr>
<tr>
<td>19</td>
<td>290254</td>
<td>21926</td>
<td>7.55</td>
</tr>
<tr>
<td>20</td>
<td>593686</td>
<td>15982</td>
<td>2.69</td>
</tr>
<tr>
<td>Avg</td>
<td>532624</td>
<td>18468</td>
<td>4.02</td>
</tr>
</tbody>
</table>

Table 4.1 & Figure 4.2: Table and figure representation of the number of waypoints before and after cleanup on several different polygon soups. Map size 1025x1025 units.

### 4.1 Research question (A)

In Table 4.1 we present the waypoint count of 20 iterations of different polygon soups. The table displays the number of waypoints before and after the waypoint cleanup takes place. This metric proves the possibility of generating waypoints from a polygon soup and it also answers research question (A). A more graphical representation can be seen in Figure 4.2.

#### 4.1.1 Waypoint Generation results

The result of every step in the algorithm to automatically generate waypoints can be found in Figure 4.1. In Figure 4.1a we see the input data in the form of a polygon soup and the size of the terrain used was 1025x1025 units which generated a terrain with over two million triangles. From the input data we detected each triangle and either defined them as blocked or unblocked. The triangle is blocked if the dot product between the normal and the up vector is below 0.8 radians, converts to 45 degrees. For every blocked triangle a waypoint is placed on every vertex as shown in Figure 4.1c. This creates waypoints which is unnecessary due to it exist waypoints close
4.2. Research question (B)

Table 4.2: Average time consumption and waypoint cleanup percentage of every cleanup step in the waypoint generation algorithm. The average is calculated from ten samples. Map size 1025x1025 units.

<table>
<thead>
<tr>
<th>Cleanup stage</th>
<th>Average time, ms</th>
<th>Waypoint removal (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocked</td>
<td>273.059</td>
<td>93.3</td>
</tr>
<tr>
<td>X-axis</td>
<td>30.948</td>
<td>1.15</td>
</tr>
<tr>
<td>Y-axis</td>
<td>17.613</td>
<td>1.0</td>
</tr>
<tr>
<td>Diagonal</td>
<td>14.430</td>
<td>0.3</td>
</tr>
<tr>
<td>Cluster</td>
<td>42.262</td>
<td>0.3</td>
</tr>
</tbody>
</table>

These are removed to greatly reduce the connection time, see Figure 4.1d. The remaining waypoints now describes edges of the blocked geometry. The waypoints are offset from the center of the triangle of origin towards the position of the vertex it is placed upon as shown in Figure 4.1e. The waypoints are connected to each other after the offset is applied and the GPU was utilized to speed up the process. A visible representation of the connection step can be seen in Figure 4.1f.

4.1.2 Results of removing unnecessary waypoints

Table 4.2 displays the average time consumption and percentage waypoint removal of all the cleanup steps. The percentage waypoint removal is based on the initially created waypoints. By removing these unnecessary waypoints we reduce the waypoint count by a total of 96% on average. The waypoint removal can remove up to almost 99% on some terrains. These are typically terrains covered by big lakes or steep mountain ranges. Table 4.1 shows how fluctuating the cleanup can be based on the terrain. Iterations with a high waypoint count are highly blocked irregular terrains and terrains with a low waypoint count are wide open terrains or terrains that are completely, or close to completely, blocked.

4.2 Research question (B)

Figure 4.3 presents the results of the survey. The diagram shows a positive trend towards high responsiveness and good pathfinding results. This proves that it is possible to execute pathfinding with acceptable results, on waypoints generated from a polygon soup.

4.2.1 Connection Reduction Results

To be able to traverse our waypoint, we used edge sparsification. We sampled ten different waypoint connection graphs generated by ten different input polygon soups and by the data gathered we make the assumption that the reduction will always be at least 85%, see Table 4.3. The reduction of connections the data represents is expected due to the connection rule checks if the new connection has a dot product towards an old connection that is above 0.9.
Chapter 4. Results

<table>
<thead>
<tr>
<th>#Connection using approximation</th>
<th>#Connection not using approximation</th>
<th>#Waypoints</th>
<th>Reduction(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>123519</td>
<td>1179677</td>
<td>12582</td>
<td>89.5</td>
</tr>
<tr>
<td>267415</td>
<td>2257953</td>
<td>28231</td>
<td>88.2</td>
</tr>
<tr>
<td>215899</td>
<td>1937807</td>
<td>22237</td>
<td>88.9</td>
</tr>
<tr>
<td>125960</td>
<td>1133538</td>
<td>13191</td>
<td>88.9</td>
</tr>
<tr>
<td>51373</td>
<td>474878</td>
<td>5828</td>
<td>89.2</td>
</tr>
<tr>
<td>245633</td>
<td>1953379</td>
<td>27612</td>
<td>87.4</td>
</tr>
<tr>
<td>128699</td>
<td>1163592</td>
<td>13898</td>
<td>88.9</td>
</tr>
<tr>
<td>190665</td>
<td>1768640</td>
<td>18855</td>
<td>89.2</td>
</tr>
<tr>
<td>203433</td>
<td>1848290</td>
<td>20875</td>
<td>89.0</td>
</tr>
<tr>
<td>121735</td>
<td>1116468</td>
<td>12874</td>
<td>89.1</td>
</tr>
</tbody>
</table>

Table 4.3: Connection reduction using an approximation of direction while connecting the waypoints. Ten samples with different input polygon soup.

4.2.2 User Study

The target group of our experiment were people who had some knowledge about video games and every participant were enrolled to either a game programming or technical artist program at Blekinge Tekniska Högskola. The test subjects were sent an email with a set of instructions and did the test from anywhere they liked, unsupervised. In the experiment the red cube controlled by the user was named Jonny. The survey had a total of 14 questions divided into five chapters; Startup, Collectible boxes, Camera, Walking and Terrain. The user were able to leave a comment at the end of every chapter if they had additional input. The only chapter of importance was the "Walking" chapter and the rest was decoy questions, to reduce the bias of the answers compared to if the users had known what was tested. We found out based on the Startup chapter that the average startup time was 23.5 sec. The pathfinding data sampled from the survey can be found in Figure 4.3. The results show that 92% of the users thought that the algorithm to calculate the paths where responsive, 77% thought that Jonny almost always walked where the users intended and 77% thought that the paths Jonny "chose" was the right path to take. Those who chose to put a low score in the pathfinding also left comments why they did so. They were annoyed that Jonny crossed the gray area as we had point out in the instructions was blocked terrain.
4.2. Research question (B)

**Figure 4.3:** Pathfinding questions data sampled from the 13 users doing the experiment. Each test subject could choose to answer the question by selecting a scale from 1–5.

Blue question scale: Not responsive(1), Very responsive(5).

Red question scale: Never(1), Always(5).

Yellow question scale: Never(1), Always(5).

Survey Results

![Survey Results Diagram]

- **How responsive were the movement?**
- **Did Jonny always walk as you intended?**
- **Did Jonny sometimes take paths that didn’t feel like the right ones?**
Chapter 5

Analysis and Discussion

5.1 Map generation

The DSA generates varying map layouts and it is a good way to test the waypoint generation algorithm in multiple different environments. Different kind of zones were added depending on the height value of the area as well as the angle of the polygon. Polygons with steep angles represent non-traversable terrain and height values below a threshold correspond to water covered areas. Maps that are very open or almost fully blocked produce few waypoints, and are therefore cheaper to create than maps that contain lots of small blocked areas. The most time consuming part of the map creation process is the waypoint connection algorithm. This stays true until the maps size becomes too big. The height map creation time escalates due to the Gaussian blur, and the time it takes to create the terrain will eventually surpass the connection algorithms time consumption. The limitation of maps of size 1025x1025 was therefore introduced. The map creation is not part of the waypoint generation algorithm and we only wanted a reliable method to generate multiple different polygon soups.

5.2 Waypoint generation

Smoothing the blocked triangles previous to waypoint cleanup improves the waypoint removal during the cleanup step. The smoothing algorithm blocks any triangle that is non-traversable by the character due to size restrictions. Several waypoints could persist along an edge after cleanup, before the smoothing was introduced. This was due to the gap check that the cleanup performs. The gap check evaluates only one triangle at a time and can therefore not make a difference of real geometry gaps and one unblocked triangle. Unnecessary waypoints would persist after cleanup along edges with triangle gaps, and smoothing the gaps was therefore necessary.

Connecting the waypoints on the CPU is not feasible for larger environments. The connection cost in Big O notation is O(n²), where n is the number of waypoints. The CPU cannot handle the exponential scaling of the algorithm good enough and connecting the waypoints on the GPU is therefore necessary. The waypoint generation algorithm can handle environments of even bigger sizes than the ones in the experiments, but it would probably be better to take the same approach as Wardhana et al.[14] and subdivide these into smaller regions and connect the regions to each other.
5.3 User perception

The results showed a positive trend towards high quality and highly responsive pathfinding. The negative feedback was mostly related to the character walking on blocked areas. The pathfinding is in fact correct, but the texture used in the experiment is generated during creation and it is not high enough resolution. This leads to that some areas get faulty texturing and are therefore presented as blocked areas during runtime. This was an oversight on our part. The texture was implemented to improve the visual effects of the environment, but in hindsight it would be better to present the actual triangle data instead. The difference of the texture and the triangle data can be seen in Figure 4.1a and Figure 4.1b.

Manipulating the path returned by the pathfinding algorithm results in smoother movement patterns, and this was desired for the experiment. The path manipulation did most certainly affect the users responses in the survey, but we wanted to prove that the graph can be used in real time game environments with acceptable results. The waypoint graph can be used without manipulation, but is then more suited for applications where movement along straight lines is desired. The sample size and the reliability of the experiment could possibly be questioned, as the margin of error is just above 25 percent, but we believe that the trend would continue since 75% of the users were positive about the games pathfinding and most of the negative feedback was related to the faulty texturing.
Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this thesis, we have presented an algorithm to generate a waypoint graph from a polygon soup. The algorithm includes waypoint reduction and edge sparsification. Our waypoint graph is well suited for a deterministic environment with agents of the same size. The user study proved that the waypoint graph can be used to provide high quality pathfinding. The pathfinding algorithm was perceived as responsive and the paths did most of the time correspond to the users expectations. The waypoint graph is less memory consuming than a grid, but a grid would most likely produce better paths without intervention during character movement. By this we have answered our thesis questions. It is possible to automatically generate waypoints to be used on random maps and it is possible to traverse automatically generated waypoints in a way that users perceive the result as realistic and responsive.

It is not feasible to find the shortest path in real-time in larger environments, even though our waypoint graph reduces the amount of connections by approximately 88% (see Table 4.3), compared to the visibility graph. Finding a path is still achievable with the real-time requirement, but it can sometimes produce unnecessary traversal and lengthy routes. The users feedback to the pathfinding was trending towards the desired outcome, i.e. the paths were responsive and qualitative. We can therefore conclude that it is possible to traverse generated waypoints with acceptable results.

6.2 Future work

Connecting the waypoints is the most time consuming part of the algorithm and for future work, the waypoint connections could be calculated using DirectX Raytracing instead of compute shader raytracing using quad tree traversal.

Another option would be to create an algorithm for real-time dynamic manipulation of the waypoint graph. The algorithm should be able to handle waypoint addition and waypoint removal. Waypoint addition should be executed when new obstacles are introduced to the environment and waypoint removal should be executed when obstacles are removed from the environment. Waypoint removal is harder to achieve, this because the graph can lose connectivity across areas where waypoints are removed.

Also support for multiple character sizes could be added to the waypoint graph. Each edge should then contain data about the maximum character size.


Appendix A

Source Code

```c
void * qtSer = malloc(TREE_SIZE); // Allocate a buffer which is the size of the tree
UINT currentOffset = 0;
for (size_t i = 0; i < qt.size(); i++)
{
    UINT sizeOfTriInd = (UINT)qt[i].TriangleIndices.size() * sizeof(UINT); // Get the size of the
    triangle indices
    memcpy((char*)qtSer + currentOffset, &qt[i], qt[i].ByteSize - sizeOfTriInd); // Copy the node
    minus the triangle indices
    currentOffset += qt[i].ByteSize - sizeOfTriInd; // offset the memory
    memcpy((char*)qtSer + currentOffset, qt[i].TriangleIndices.data(), sizeOfTriInd); // copy the
    triangles
    currentOffset += sizeOfTriInd; // offset the memory
}
```

Listing A.1: Code used to serialize the quad tree data
//Structures used to define connections between waypoints
struct TreeNode // This define a node in the quadtree
{
    uint  ByteStart;  // Where in the memory the node is located
    float2 Min;
    float2 Max;
    uint  NrOfChildren;
    uint  ChildrenByteAddress[4];  // Where the children of the node are located in the memory
    uint  NrOfTriangles;  // The number of triangles the node contains
};

struct AddressStackElement // Stack element used to store traversed tree nodes
{
    uint  Address;  // Address of the node
    uint  TargetChildren;  // Which children to target next
};

struct Waypoint // Definition of a GPU waypoint
{
    uint  Key;  // Used to access the waypoint on the cpu
    uint  NrOfConnections;
    float2  Pos;
    uint  Connections[256];  // stores the key values the connected waypoints
};

cbuffer OffsetBuffer : register(b0) // An offset buffer used to offset the
{  // groupID input to the compute shader
    uint4 OFFSET;
}

StructuredBuffer<Triangle> Triangles : register(t0);  // Triangles
ByteAddressBuffer QuadTreeBuffer : register(t1);  // Quad Tree
RWStructuredBuffer<Waypoint> Waypoints : register(u0);  // Waypoints

Listing A.2: Structures used in the traversal algorithm on the GPU.

TreeNode GetNode(in uint address, out uint triangleIndexAddress)
{
    TreeNode node = (TreeNode)0;
    address += 4;  // Skip the total byte size of the node
    node.ByteStart = QuadTreeBuffer.Load(address);  // Load the start address
    address += 4;  // offset the address to target next
    node.Min = asfloat(QuadTreeBuffer.Load2(address));  // Load the Min value
    address += 8;
    node.Max = asfloat(QuadTreeBuffer.Load2(address));  // Load the Max value
    address += 8;
    address += 4;  // Skip the node depth
    node.NrOfChildren = QuadTreeBuffer.Load(address);  // Load number of children (can be 4 or 0)
    address += 4;
    address += 4 * 4;  // childrenIndices
    [unroll]
    for (uint j = 0; j < 4; j++)
    {
        node.ChildrenByteAddress[j] = QuadTreeBuffer.Load(address);
        address += 4;  // childrenByteAddress[i]
    }

    node.NrOfTriangles = QuadTreeBuffer.Load(address);  // Load triangle count
    address += 4;
    triangleIndexAddress = address;  // This address represents the containing triangle memory start
    return node;
}

Listing A.3: Function to extract a tree node from the byte address buffer.
size_t increment = 512; // Thread groups to dispatch
// Add extra iterations so nothing is missed, if outside memory, the shader discards the execution
size_t gpuwpSize = gpuWp.size() + increment - 1;

int error = 0;
for (size_t i = 0; i < gpuwpSize; i += increment)
{
    error = 0;
    // Offset the GPU Values
    D3D11_MAPPED_SUBRESOURCE offsetData;
    deviceContext->Map(offsetBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &offsetData);
    memcpy(offsetData.pData, &DispatchOffset, sizeof(DirectX::XMUINT4));
    deviceContext->Unmap(offsetBuffer, 0);
    deviceContext->CSSetConstantBuffers(0, 1, &offsetBuffer);
    // Dispatch
    deviceContext->Dispatch(increment, 1, 1);
    DispatchOffset.x += increment;
    deviceContext->End(async);
    UINT queryData;
    while(S_OK != deviceContext->GetData(async, &queryData, sizeof(UINT), 0)) // Spinlock to
    freeze the CPU while the GPU is not done
    {
        error++;
    }
}

Listing A.4: Code used to serialize batch the dispatches of work to the GPU
static const uint WP_SPLIT = 1024;

[numthreads(WP_SPLIT, 1, 1)]
void main(uint3 groupID : SV_GroupID, uint3 groupThreadID : SV_GroupThreadID)
{
    uint waypointTarget = groupID.x + OFFSET.x; // Define the waypoint this thread should find
    Waypoint target = Waypoints[waypointTarget];
    
    uint wpSplit = groupThreadID.x; // The current threads group id is used to split what waypoint
    
    uint nrOfWaypoints = 0, dummy = 0;
    Waypoints.GetDimensions(nrOfWaypoints, dummy); // Get the total number of waypoints on this
    terrain
    
    uint numberOfWP = max(nrOfWaypoints / WP_SPLIT, 1); // Counts how many waypoints each thread
    group should process
    uint wpStart = numberOfWP * wpSplit + waypointTarget + 1; // Which waypoint each thread in each
    group should start on
    uint wpEnd = wpStart + numberOfWP; // Which waypoint each thread in each group should end on
    wpEnd = min(wpEnd, nrOfWaypoints); // This is a safe so we dont read outside our waypoint buffer
    
    float2 origin = target.Pos; // Line start
    for (uint i = wpStart; i < wpEnd; i++) // For every waypoint this thread should be processing
    {
        bool hit = false;
        Waypoint towards = Waypoints[i];
        float2 end = towards.Pos; // Line end
        
        float l = length(end - origin);
        if (l > 128) // If the length between the waypoints are above 128, there is no connection
            continue;
        
        AddressStackElement nodeStack[7]; // The stack only need to be as big as the quad tree
        because of Depth first
        uint nodeStackSize = 0;
        uint triIndexAddress = 0;
        TreeNode node = GetNode(0, triIndexAddress); // Get the root node, this will always be at 0
        in the buffer
        
        bool intersection = false;
        
        // Add the start node to the stack
        nodeStack[nodeStackSize].Address = node.ByteStart;
        nodeStack[nodeStackSize].TargetChildren = 0;
        nodeStackSize++;;
    }
}

Listing A.5: HSLS Code used to setup the current thread for traversal
while (nodeStackSize > 0 && !intersection) // While there are nodes in the node stack and no intersection have been found
{
    uint currentNode = nodeStackSize - 1;
    node = GetNode(nodeStack[currentNode].Address, triIndexAddress);
    if (nodeStack[currentNode].TargetChildren < node.NrOfChildren) // If not all children have been traversed
    {
        TreeNode child = GetNode(
            node.ChildrenByteAddress[nodeStack[currentNode].TargetChildren++],
            triIndexAddress);
        if (LineQuadIntersect(origin, end, child.Min, child.Max)) // If the line intersects with this tree node
        {
            if (child.NrOfTriangles > 0) // if this node has triangles
            {
                for (uint triIt = 0; triIt < child.NrOfTriangles && !intersection; triIt++) // For all triangles
                {
                    Triangle tri = GetTriangle(triIndexAddress, triIt);
                    intersection = LineTriangleIntersect(origin, end, tri); // Check for intersection
                    if (intersection)
                    {
                        hit = true; // There exists an intersection between the line and a triangle
                        break;
                    }
                }
            }
            else // If the node don't have any triangles add this node to the node stack
            {
                nodeStack[nodeStackSize].Address = child.ByteStart;
                nodeStack[nodeStackSize].TargetChildren = 0;
                nodeStackSize++;
            }
        }
        else
        {
            nodeStackSize--; // pop the current node
        }
    }
    else
    {
        nodeStackSize--; // pop the current node
    }
} // While End

Listing A.6: HLSL Code used to traverse the quad tree and find connections of waypoints

if (!hit) // If no triangle were hit, create a connection between these two waypoints
{
    uint index;
    InterlockedAdd(Waypoints[waypointTarget].NrOfConnections, 1, index);
    index = min(index, 255);
    Waypoints[waypointTarget].Connections[index] = towards.Key;
} // For end
} // Main end

Listing A.7: HLSL Code used to store a connection to a waypoint
Appendix B

Supplemental Information

Instructions:

Please read the instructions carefully before executing the experiment, thank you!

- It is very important that you have a graphic card in your computer while trying to run this application, else it will not be able to run.
- If for some reason you cannot run the program at all, thank you anyway for trying to participate.
- You downloaded the program [here](https://drive.google.com/file/d/1ypWJw7XXoTEabVI2HyEqymS2EoDm5nFl/view?usp=sharing)
- Thank you for your participation in our experiment!

Summary

- Download the program ([download](https://drive.google.com/file/d/1ypWJw7XXoTEabVI2HyEqymS2EoDm5nFl/view?usp=sharing))
- Startup the game (StartHere.exe)
- Write down the total loading time
- Help Jonny to collect the boxes (Use mouse to move the camera, rotate the cam, and give Jonny a location to go to).
- Jonny can only walk on grass and snow.
- When you are done, answer the survey.

Introduction

After you have played the game, please answer this survey ([https://forms.gle/Nr9zABcUgG17RiNF8](https://forms.gle/Nr9zABcUgG17RiNF8)).

Note: There is a chance that the loading will freeze forever. If it takes above 30 seconds after the right black window prints "Placing Coins", please restart the program.

To close the program during startup, close the console window.

When the game starts, the console will print the total loading time. Please write this down, this you will be asked to enter in the survey.

When you have collected all the rotating boxes on the map, the game will freeze and restart. You are welcome to play the game more than once.

Objectives

The goal is for you to guide Jonny to his lost rotating yellow boxes.

When you right click the terrain, Jonny will follow your guidance, if there exist a path to the selected point.

This is very important, Jonny can only walk on snow and grass and cannot walk on stone (the dark gray area on the terrain).

When you have collected all the rotating yellow boxes the game will restart. You are welcome to continue playing.

When you are done playing, please write down the total loading time from the right black window. Exit the game and answer the survey.

Figure B.1: Instructions used in the user experiment
Help Jonny Collect the Boxes - Survey

In this survey you will be asked to write down your thoughts of the game you just played.

*Obligatorisk

**Startup questions**

1. Did the program freeze during startup? *
   - Never
   - One Time
   - Two Times
   - Three Times
   - Above Three Times

2. How long did it take for the program to start? *(This is the number you wrote down)*

3. What GPU are you using? *

4. What CPU are you using? *

5. Any comments on the startup process?

**Collectible boxes**

6. Was it easy to locate the collectible boxes? *
   - I never found the boxes
   - 1
   - 2
   - 3
   - 4
   - 5
   - I always found the boxes

**Camera**

7. Did the camera movement feel natural to you? *
   - I do not agree
   - 1
   - 2
   - 3
   - 4
   - 5
   - I totally agree

8. Did the camera rotation feel natural to you? *
   - I do not agree
   - 1
   - 2
   - 3
   - 4
   - 5
   - I totally agree

9. Did the camera zoom feel natural to you? *
   - I do not agree
   - 1
   - 2
   - 3
   - 4
   - 5
   - I totally agree

10. Any comments on the camera?

**Walking**

11. How responsive were the movement? *
    - Not responsive
    - Very responsive

12. Did Jonny always walk as you intended? *
    - Never
    - 1
    - 2
    - 3
    - 4
    - 5
    - Always

13. Did Jonny sometimes take paths that didn’t feel like the right ones? *
    - Never
    - 1
    - 2
    - 3
    - 4
    - 5
    - Always

14. Any comments on the movement?

**Terrain**

15. Did you enjoy the automatically generated terrain? *
    - Not at all
    - Very beautiful

16. Did the terrain shading (lighting) feel natural to you? *
    - I do not agree
    - 1
    - 2
    - 3
    - 4
    - 5
    - I totally agree

17. Any comments on the terrain?

Figure B.2:
Survey used during the user experiment

https://docs.google.com/forms/d/1fpYvM61fugMr22W9B8FI1o1K5lhUrxBu3nu17HCgG2c/edit