Master of Science in Engineering: Game and Software Engineering
May 2024

GIPP: GPU-based Path Planning and Navigation Mesh Generation

A Novel Automatic Navigation Mesh Generator and Path Planning Algorithm using the Rendering Pipeline

Elliot Lundin
Felix Mathiasson

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 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):
Elliot Lundin
E-mail: ellu19@student.bth.se

Felix Mathiasson
E-mail: fema18@student.bth.se

University advisor:
Dr. Prashant Goswami (Docent)
Department of Computer Science (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

Background. Geometry-Independent Path Planning (GIPP) can be done by generating a navigation mesh and computing paths on that mesh in real-time for parallel and dynamic path planning. However, many of the existing algorithms are not suitable for the Graphics card, therefore a new path planning algorithm is created. Hardware Accelerated Line Of Sight (HALOS) performs parallel path planning on grid maps in real-time using the GPU.
Objectives. This thesis aims to implement an automatic navigation mesh generation algorithm using the GPU rendering pipeline and a GPU-bound path planning algorithm for a grid-based map. The proposed method should generate accurate paths and run in real-time. To gather results, the methods are measured in run-time on different types of hardware and scenarios.
Methods. Multiple experiments are conducted. A navigation mesh is generated in real-time using the rendering pipeline of the GPU. In addition, a novel path planning algorithm generates paths in real-time using the GPU by utilizing line of sight on the navigation mesh. GIPP is a multi-source, single-destination algorithm. The path planning is done parallel and dynamically to navigate around moving obstacles.
Results. The experiments show that GIPP can generate a dynamic navigation mesh in real-time. However, the coverage of GIPP is poor, and some resolutions of GIPP result in agents being unable to reach the goal node. The performance effect of dynamic worlds on path planning is not noticeable compared to static worlds.
Conclusions. The proposed method can perform real-time navigation mesh generation and path planning. Different resolutions of GINT show inconsistencies in the length of the path generated. This method, GIPP, is well suited for complex, dynamic, single-floor meshes that more traditional navigation mesh generators are not guaranteed to handle in real-time. The main performance bottleneck for GIPP is the number of layers created during path planning.

Keywords: Geometry-Independent, Navigation, Mesh Generation, Path planning, Pathfinding, Rendering.

Sammanfattning

Bakgrund. Geometrioberoende vägplanering (GIPP) kan utföras genom att generera ett navigationsnät och beräkna vägar på detta nät i realtid för parallell och dynamisk vägplanering. Många vägplaneringsalgoritmer kan inte köras i realtid på grafikkortet. Därför har Hardware Accelerated Line Of Sight (HALOS) skapats, vilket utför parallell vägplanering i realtid med hjälp av GPU:n.
Mål. Denna avhandling syftar till att implementera en automatisk algoritm för generering av navigationsnät med hjälp av GPU:ns renderingspipeline och implementera en GPU-bunden vägplaneringsalgoritm för en rutbaserad karta. Den föreslagna metoden genererar vägar och körs i realtid. För att samla in resultat mäts metoderna i körtid på olika typer av hårdvara och scenarier.
Metoder. Flertalet experiment utförst på GIPP. Ett navigationsnät genereras i realtid med hjälp av GPU:ns renderingspipeline och en ny vägplaneringsalgoritm genererar vägar i realtid med hjälp av sikten längs navigationsnätet. Denna algoritm har flera källor med en destination (MSSD) för att hantera ett stort antal agenter. Vägplaneringen görs parallellt och dynamiskt för att navigera runt rörliga hinder.
Resultat. Experimenten visar att GIPP kan generera ett navigationsnät i realtid. GIPP har dock dålig precision när det gäller att generera effektiva vägar mot målet. Vissa upplösningar leder till att agenter inte når slutmålet. Dynamiska världar har omärkbar påverkan på prestandan i jämförelse med statiska världar när det gäller vägplanering.
Slutsatser. Den föreslagna metoden kan utföra navigationsnätsgenerering och vägplanering i realtid. Olika upplösningar av navigationsnätet visar att vägplanering har olikeheter i avstånd. Denna metod, GIPP, lämpar sig väl för komplexa, dynamiska, enplansvärldar. GIPPs flaskhals i prestandan är mängden lager som skapas under vägplaneringen.

Nyckelord: Geometrioberoende, Navigering, Mesh generering, Vägplanering, Pathfinding, Rendering.

Acknowledgments

We express our gratitude to Dr. Prashant Goswami, our supervisor, for his dedication and support throughout the completion of this master thesis.

We would also like to thank Veronica Sundstedt for her assistance in providing us with an office for the duration of this master thesis.

Contents

Abstract i

Sammanfattning iii

Acknowledgments v

1 Introduction 1
1.1 Geometry-Independent Path Planning . . . 3
1.2 Aims and Objectives . . . 3
1.3 Research Questions . . . 4
1.4 Scope and Limitations . . . 4
1.5 Ethical, societal and sustainability aspects . . . 4
1.6 Outline . . . 5

2 Background 7
2.1 Terminology . . . 7
2.1.1 Shared terminology . . . 7
2.1.2 GINT terminology . . . 8
2.1.3 HALOS terminology . . . 8
2.2 Geometry-Independent Navigation Texture . . . 9
2.3 Hardware Accelerated Line Of Sight . . . 10
2.3.1 HALOS Layers . . . 11

3 Related Work 13
3.1 Representations for Path Planning . . . 13
3.2 Navigation Mesh . . . 13
3.3 Navigation Mesh Generation . . . 14
3.4 Path Planning Algorithms . . . 15
3.5 Geometry-Independent Techniques: Shadow Mapping . . . 16
3.6 Multiple Sources Single Destination . . . 17

4 Method 19
4.1 Implementation . . . 20
4.1.1 ImGui . . . 20
4.1.2 GINT . . . 20
4.1.3 HALOS . . . 23
4.2 Parallelization and the Sequential Parts of GIPP . . . 25
4.3 Precision . . . 26

4.4 Performance 29
4.5 Auxiliary Data 29

5 Results and Analysis **31**
5.1 Performance Results 34
5.2 Precision Results 39
5.3 Auxiliary Data 42

6 Discussion **47**
6.1 Performance 47
6.1.1 Memory 47
6.1.2 Comparison 48
6.1.3 Issues with the implementation 49
6.2 Precision 49
6.3 GIPP Usability 50
6.3.1 Advantages and Disadvantages of GIPP 50

7 Conclusions and Future Work **53**
7.1 The Performance of GIPP 53
7.2 The Precision of GIPP 53
7.3 Future Work 53
7.3.1 Optimization 54
7.3.2 Multiple GINTs 54
7.3.3 MSMD 54
7.3.4 Other Rendering Methods 54
7.3.5 Reducing Overhead 55
7.3.6 Overall Specialization Route 55

References **57**

A Supplemental Information **59**

Chapter 1
Introduction

Video games are well-known in society and a big part of many people's lives. The purpose of video games is entertainment, whether that is escaping from reality by playing an elf in a fantasy role-playing game or playing for competition and trying to dominate foes, or simply spending time with friends online.

For a game to be interesting and less stale, non-playable characters, otherwise known as NPCs, should be able to navigate in the game world. To accomplish this, algorithms are used to calculate efficient paths that the NPCs can navigate with. The most popular algorithm is A*, pronounced A-star, which is an efficient algorithm for finding the shortest paths between two points fast. Another method is a flood fill algorithm which generates a heat map to be used with vector kernel convolution to generate a vector field from which agents can navigate the game world. The flood fill algorithm starts in a node and explores its neighbors to find paths. This can be compared to flooding a maze with water. Along each step, it counts the distance to generate a heatmap. Vector kernel convolution takes information from the node's eight neighbors to generate a vector for the node pointing to the most optimal path. A problem with the most common modern pathfinding algorithms is that as the number of NPCs using pathfinding goes up, the time it takes for the algorithms to finish goes up as each run of the algorithm calculates a path from one point to another and each additional NPC would require another run of the algorithm. This is where parallel pathfinding becomes relevant, instead of calculating the path for each NPC one at a time, it saves a lot of time to calculate them all at the same time, at which point it takes as much time to calculate the path for one NPC as it takes to calculate the path for fifty NPCs.

Figure 1.1: This picture shows gameplay footage of the game Starcraft 2, a real-time strategy game where players expand their bases and create units to fight the other player with. In the later stages in a game of Starcraft 2, players control many individual units to the point that the program needs to handle hundreds of agents utilizing path planning, which can be a bottleneck.

For the pathfinding algorithms to work, the game world must be represented by data. The navigation mesh is usually a 2D simplification of a 3D world with the idea that the existing 2D search algorithms become able to efficiently make use of the simplified representation to plan paths in a 3D world. The navigation mesh makes use of the fact that most game environments have easily discernible "walkable" and inaccessible areas. The navigation mesh is made out of convex polygons, a convex shape does not have any corners that have an inner angle less than 180 degrees. Importantly, this means that any straight line can be made between any two points in the convex polygon without an edge intersection with itself. The advantage of this fact lets path planning algorithms move in straight lines within the convex shapes that make up navigation meshes.

Unfortunately, the navigation mesh itself needs to be created. For simpler game worlds, the mesh can be created manually. When the world becomes complex, or simply too large, then manual creation and editing of the navigation mesh becomes a non-viable option. This is where navigation mesh generation comes into use. The repetitive work of creating a complex or large navigation mesh is well suited for a computer, especially so if the world itself already is represented as digital data in the same computer.

There is one issue the current fast navigation mesh generation algorithms have in common. The issue is that the navigation mesh generation algorithms require the geometry to be divided up into convex shapes. Convex shapes can generally be calculated with methods that are faster than more geometry-independent shapes, such as concave shapes. Concave shapes can be divided into multiple convex shapes, however, it usually implies that either more memory is required to keep track of more shapes overall or that the resulting pathfinding algorithm needs to search through more meshes, thus reducing its performance.

1.1 Geometry-Independent Path Planning

Geometry-Independent Path Planning (GIPP) attempts to deal with the increasing complexity of game worlds by making use of the same hardware that renders them, the Graphics Processing Unit (GPU). GIPP attempts to fill in the gaps in real-time automatic navigation mesh generation techniques by providing a geometry-independent method for creating a two-dimensional grid map of a scene. GIPP also fills in the gaps in hardware-accelerated path-planning algorithms by providing a path planning algorithm that can navigate a grid map in real-time via the GPU. These two parts, the automatic navigation mesh generation technique, and the path planning algorithm, make up GIPP. In addition, two new data structures have been made, to fill the gaps in real-time navigation mesh generation on the GPU and path planning on the GPU.

1.2 Aims and Objectives

This thesis explores GIPP as an alternative method of automatic navigation mesh generation and path planning on the GPU to achieve real-time path planning on complex dynamic game worlds using hardware acceleration. Since GIPP is both an automatic navigation mesh generator and a path planning algorithm, this thesis consequently explores navigation mesh generation and path planning simultaneously. This thesis has three objectives for GIPP:

- Implementing an automatic navigation mesh generation algorithm using the GPU rendering pipeline.
- Implementing a GPU-bound path planning algorithm for a grid-based map.
- Measure and test the performance and precision to determine the real-time viability of the two algorithms.

1.3 Research Questions

The objectives bring up interesting aspects of this thesis. The first is on whether the resolution of the grid map generated by the navigation mesh generator in GIPP affects the path planning algorithm in any significant way. The second is if GIPP performs well enough to achieve real-time speed for automatic navigation mesh generation and path planning for a game world. Lastly if a dynamic world, a game world in which the obstacles have varying positions over time, affects the performance of GIPP is considered. For the research questions, real-time is defined as one iteration taking less than 16.67 milliseconds. Together these aspects make up two research questions:

RQ1. How do dynamic environments affect the **performance**, in terms of execution time and memory, and what is the real-time viability of GIPP?

RQ2. How does the resolution of GIPP affect the **precision** of finding the shortest path?

1.4 Scope and Limitations

The scope of the thesis contains only tests on a single navigation mesh, therefore no scenes with multiple floors are used to answer the research questions. The navigation mesh generation part of GIPP is simplistic and with little optimization, to test how a simple version of GIPP performs in a real-time environment. Testing for the path planning path of GIPP is done using only a single goal with a set number of agents.

GIPP is mainly performed on the GPU, meaning that only overhead and certain output data occur on the CPU, therefore any time measurements are taken on the GPU rather than the CPU. The hardware acceleration and rendering pipeline library used for this thesis is DirectX11, this also implies that the tests can only be done on the operating system Windows, in this case, the operating system used is both Windows 10 and Windows 11. C++ using the MSVC compiler is used for code on the CPU. Unlike most previous navigation mesh generation research articles, this thesis makes experiments on performance on multiple computer setups, which are specified in Results 5.

1.5 Ethical, societal and sustainability aspects

There is no inclusion of personal data inclusion with either GIPP or the testing method. One relevant ethical issue is the intellectual property of GIPP. GIPP is made by the same people who wrote this thesis and is not open source. GIPP, by technicality, is owned by BTH. No additional libraries used for this project make use of data collection or restricting licenses. The testing scenarios include publicly available game worlds. There is no AI training nor is there machine learning and therefore no training data is used for this thesis. User data is also not used.

Games often act as a medium for software innovation, in this case, the generation of navigation textures can be remade into maps for navigation in areas that can

be rendered in 3D, such as a floor layout of a building. This can for example be repurposed into a way to find the shortest path to a fire exit on a building floor. However, since GIPP requires a model for the navigation texture to be rendered via the render pipeline, a 3D model has to be made of whatever it needs; a flat satellite image does not work as intended for GIPP.

The GPU is generally more energy efficient than the CPU for simple instructions [14], which the GIPP is mostly made up of. This means that it has a positive energy sustainability aspect. GIPP is therefore potentially more energy efficient than its CPU navigation mesh generation and CPU pathfinding counterparts. However, when looking at the greater picture, the energy sustainability aspect likely does not matter since the CPU, instead of idling, performs other instructions in the meantime.

1.6 Outline

The background provides terminology and information on GIPP's two parts, GINT and HALOS. After the background is a chapter about related work regarding navigation meshes, navigation mesh generation, and path planning. In addition, there are also related works that have a connection to GIPP's technical aspects. The method contains the GIPP algorithm and how it is implemented to answer the questions of this thesis. The results of the experiments are visualized in the results, which are split into three parts: the performance of GIPP, the precision of GIPP, and lastly auxiliary data, whose purpose is to give context for comparisons with other works. The discussion analyzes and reflects on the results concerning the thesis questions, aims, and objectives. Lastly, conclusions have been made in addition to future work for GIPP. Additional resources can be found in the appendix, named Supplemental Information.

Chapter 2

Background

Geometry-Independent Path Planning (GIPP) is both an automatic navigation mesh generator and a path planning algorithm, of which both parts are calculated on the GPU instead of the CPU. The idea is that the rendering pipeline handles the navigation mesh generation whilst the General-Purpose-GPU (GP-GPU) does the path planning, therefore avoiding communication and potential memory bandwidth issues between the CPU and the GPU. The automatic navigation mesh generator part worked out well and thus became Geometry-Independent Navigation Texture (GINT). On the other hand, converting a CPU-bound path planning algorithm to the GP-GPU proved not to be feasible for real-time performance due to the GPU's slow performance for complex instructions and synchronization compared to the CPU. Therefore HALOS is intentionally a simple path planning algorithm that makes use of parallelization on the GPU to achieve real-time performance.

This thesis intends to implement a novel automatic navigation mesh generator and a GP-GPU bound path planning algorithm to further the knowledge of alternative ways of path planning using the GPU. GIPP aims to fill the gap in research of using the rendering pipeline and the GP-GPU for navigational purposes.

2.1 Terminology

Since this thesis contains the implementation of two novel algorithms, the terminology must be well explained.

2.1.1 Shared terminology

These are words or phrases that are used by both algorithms.

- GIPP - Geometry-Independent Path Planning. This is the name for GINT and HALOS as a whole.
- Pathfinding - A synonym to Path Planning.
- Pixel Shader - This is the DirectX 11 equivalent to the Fragment Shader.
- Dispatch - A Dispatch is a Compute Shader call with a variable amount of thread groups.
- Compute - Shader - An HLSL file that runs on the GP-GPU.

- GP-GPU General Purpose Graphics Processing Unit. This is the part of the GPU that runs the compute shaders.
- Map - A map is a two-dimensional texture that contains the data of either GINT or HALOS.
- Cost - Each tile has a cost. The cost is the length of traversal whenever it is greater than zero.
- Tile - A tile is a pixel on a Map that either contains that for the GINT Map or the HALOS Map.

2.1.2 GINT terminology

These are words or phrases that are unique to GINT that HALOS does not make use of, should it be decoupled from GINT.

- GINT - Geometry-Independent Navigation Texture.
- GINT Pipeline - The rendering pipeline that renders meshes to the GINT Map.
- Obstacle - A tile with a type of 0. This is considered an inaccessible part of the map.

2.1.3 HALOS terminology

These are the terminology and phrases that GINT does not make use of.

- HALOS - Hardware Accelerated Line Of Sight.
- MSSD - Multiple Sources, Single Destination types algorithms find paths from several points in the world to a single point in the world.
- SSSD - Single Source, Single Destination types of algorithms find a path between two points in the world.
- Agent - An agent utilizes path planning in order to navigate in the world.
- Goal - The tile that contains the destination point. This is also referred to as the goal node.
- Layer - A value from 0 and up. A set of tiles with a direct line of sight to a tile with a layer value of exactly one less. The N:th layer has a line of sight to the N-1:th layer. The goal node has a Layer type of zero.
- Layer type - An explored tile has a layer type value corresponding to the Layer it belongs to.
- Border Tile - A border tile is a tile with a layer type that borders one or more tiles that are unexplored.

2.2 Geometry-Independent Navigation Texture

Geometry-Independent Navigation Texture (GINT) has its origin, although in a limited form, from earlier projects. A picture of GINT in one of the previous projects can be found in A.7. The original purpose of GINT in these earlier projects was to create a grid collision detection map of a static world map at which characters can find out what they are walking on from just a pixel on the collision detection map. The advantage this method had over a more traditional method, the traditional way of making use of ray casting, is that the layout of the vertices does not affect the performance of GINT. The idea came from a time limitation and the fact that the game world model was not a height map, meaning that finding the correct triangle to ray cast on is tricky, and potentially very slow. GINT proved to be useful in reducing the development time of the project by allowing the developers of the project to let the GPU create the navigation mesh for the game world automatically instead, an advantage shared by other navigation mesh generation algorithms.

Additionally the navigation mesh is not meant to be seen by the player. It is possible to have invisible objects for navigation in GINT as well by rendering the objects in GINT but forgoing rendering them for the visual pipeline of the game, thus it is possible to make use of simplifications, should one need it. However, this brings us back to the problem of having to do additional work for the navigation mesh, which is what GINT is intended to avoid.

GINT, however, can be used for another purpose than collision detection. For this thesis, GINT makes use of the map it creates for pathfinding, thus making GINT a data structure for the path planning algorithm to work on. As mentioned, the pathfinding algorithm is on the GP-GPU instead of the CPU. The reason pathfinding is being done on the GP-GPU is because GINT generates the map using the graphics pipeline, thus being on the GPU first. By having pathfinding, the only algorithm to make use of GINT in the scope of this thesis, be done on the GPU as well then calls and memory copying, which are very slow between the CPU and GPU, can be avoided. It is not possible to avoid all calls and memory copying, thus the remaining calls and memory copies need to be as few and small as possible. Communication between the CPU and GPU therefore limits which pathfinding algorithms can be used as they must be parallelized in order for the GPU to be more efficient than the CPU. In other words, a pathfinding algorithm with high throughput is preferable to a CPU-bound algorithm.

2.3 Hardware Accelerated Line Of Sight

Hardware Accelerated Line Of Sight (HALOS) is a path planning algorithm made specifically with GINT in mind. Its origin comes from failed attempts to make hardware accelerated path planning algorithms. The algorithm is a complete rethinking of how path planning usually works. Instead of smart ways to avoid additional operations, HALOS embraces the high thread count of the GPU by being a high throughput algorithm. Its high throughput comes in the form of having threads do parallel work that avoids synchronization as much as possible. HALOS works by having layers of line of sight toward an end goal. The first layer consists of all grids with a direct line of sight, the second layer of all grids that have a direct line of sight to the grids of layer one; layer three consists of grids with a direct line of sight on the second layers grids and so on. Each grid contains the distance from its position to the closest grid of the layer below it. This creates a heat map of distances which can later be used to create a directional map for path planning. In other words, HALOS generates a distance map from a goal node to each source (each grid) by using subsequent layers of line of sight that eventually reach all sources.

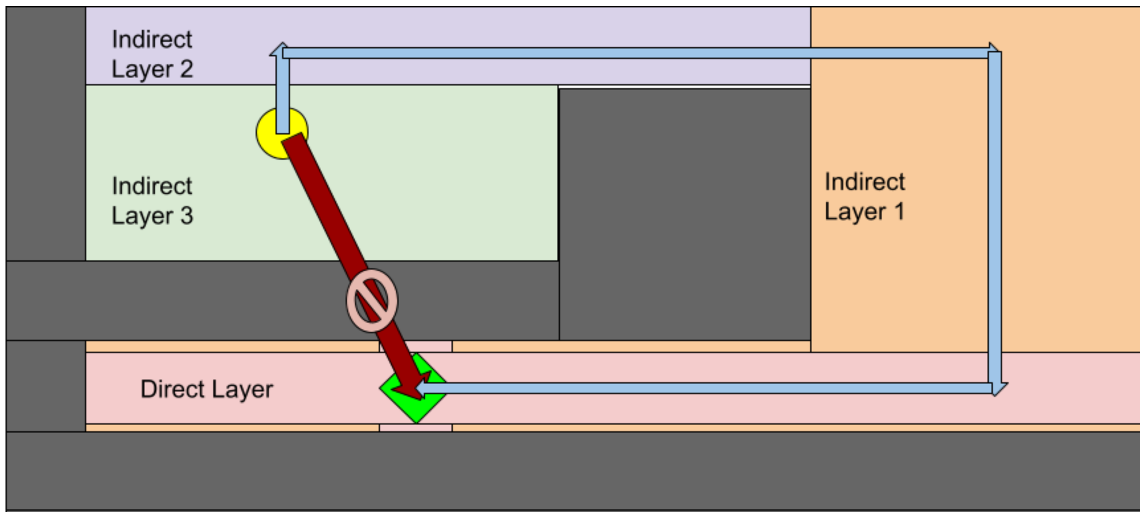

Figure 2.1: The theory of how layers of direct line of sight can provide a path from a start, the yellow circle, to a goal, the green rhombus. By moving from a layer to a less indirect layer, one eventually ends up at the goal without colliding with the dark grey walls.

One important note about HALOS is that its algorithm has an innate support for Multiple Sources Multiple Destination (MSMD), however, the implementation of HALOS for this thesis does not make use of MSMD. Another note about the map HALOS utilizes, GINT, is that GINT has potential support for non-vertices-based models, such as ray-marching metaballs. This makes it possible for GINT to create navigation meshes of fractals, so long as the rendering method can be rasterized to a texture. The texture can then be used as a grid map for HALOS. The implementation of GINT in this thesis does not have support for ray-marching. The generation of new layers can be interpreted as being MSMD since each border tile can be considered a goal in a recursive exploration algorithm.

2.3.1 HALOS Layers

A layer, in HALOS, is defined as one or more areas whose nodes (grids) have a straight path to a grid of exactly one lower layer value, a visualization of this is provided in figure 2.1. Layers are used when generating the HALOS distance map but are not used for giving directions to the agents. The purpose of layers is therefore only for the generation of the HALOS distance map. Layers provide a parallelized solution to exploring a grid-based map using the GP-GPU whose sequential part is the generation of additional layers. An example of HALOS generating a map through layers is provided in figure 2.2

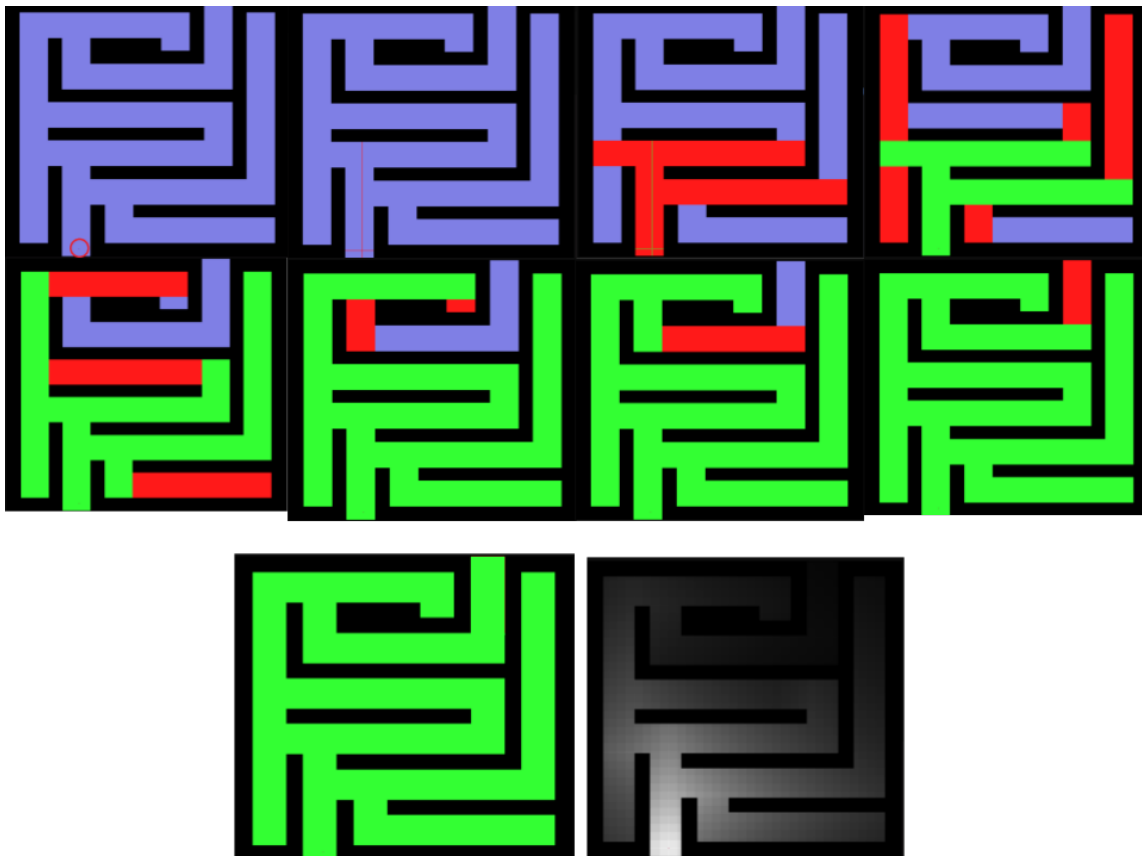

Figure 2.2: The first eight maps visualize the exploration of new layers for the generation of the HALOS distance map. The upper leftmost map is the unexplored map, in which the goal node is shown using a red outline in a circle shape. Each newly generated layer is shown as red areas and each finished layer as green. The results of the exploration can be seen in the bottom two maps. The bottom left map is the full exploration of the map and the bottom right map is a visualization of the generated distance map.

Chapter 3

Related Work

3.1 Representations for Path Planning

There are different ways of representing areas valid for path planning. these can be reduced to two main approaches of representations for path planning. One way of representing areas for path planning is by having a network of lines, or curves, that act as a graph that a search algorithm can create a path out of. The Roadmap, a set of collision-free vertices [1], usually implies difficulties with dynamic obstacles due to the representation not being able to include where obstacles are; there can be multiple different obstacle layouts that result in the same Roadmap.

The other way is by having a set of polygonal shapes that make up a navigation mesh. Van Toll et al, [18] describe different ways of structuring polygonal shapes in a navigation mesh: Triangles, Squares, Convex Polygons, and disks are some of the shapes that a search algorithm can effectively plan a path on. Importantly, the shapes are polygonal to guarantee that a character can move from any position to another within the shape without the risk of colliding with an obstacle. In GIPP's case, the pixels of the navigation mesh texture can be seen as a grid map, a set of equally sized squares, thus GINT takes the latter approach to path planning world representations; GINT therefore uses a set of polygonal shapes for its navigation mesh.

3.2 Navigation Mesh

The first to explicitly use the term "navigation mesh" was Snook in the year 2000 [16]. Navigation meshes could be treated as a way of having a grid system for 3D spaces, otherwise only seen in 2D tile-based games. In GINT's case, the navigation mesh goes full circle and becomes a 2D grid system. Snook's navigation mesh, however, assumed that it was only made out of triangles, modern navigation meshes include other convex shapes as well. The convex shape can be treated as a cell and its edges as portals, should their edges collide with other convex shapes. This is therefore known as a Cell-and-Portal-Graph (CPG). CPGs can be traversed by path planning algorithms, one popular path planning algorithm being A*.

There are different ways of structuring navigation meshes. Van Toll et al. [18] also describe various navigation meshes and their comparative purposes. One way of structuring a navigation mesh, similarly to GIPP, is by having a *Voxel-based* structure, which can afterward be made into a navigation mesh through various navigation mesh generation techniques. Voxelization can handle any geometry by

approximating it into a set of volumetric boxes. This also simplifies the process of making a navigable map; there is no need to adapt the geometry for the navigation mesh. However the approximations mean that the navigation mesh is not an exact representation of the geometry, instead the precision is based on the resolution of the navigation mesh.

Another common navigation structure is a grid map. Grid maps are simple to understand and simple to create, however, the density in the number of nodes makes them inefficient for path planning algorithms such as A*. Sturtevant [17], provides a standardized way of benchmarking grid-based path planning. These benchmarks will not be used for the experimentation of GIPP since GIPP creates its navigation meshes instead.

One way to optimize a grid map is using non-uniform grids. García et al [6], shows that the GPU is capable of performing multi-agent path planning in dynamic environments by computing the paths on the GPU but repairing the navigation mesh on the CPU. The grid map is a navigation mesh consisting of subdivisions of quad-trees. This navigation mesh structure resulted in better memory usage compared to navigation meshes consisting of uniform grids.

3.3 Navigation Mesh Generation

There are several methods of navigation mesh generation. One method is to create a navigation mesh by hand. Creating a navigation mesh by hand is time-consuming and error-prone. On the other hand, having full control of the navigation mesh means that complex pathing routes such as jumping over obstacles or interacting with objects like levers can be baked into the navigation mesh. However, creating navigation meshes by hand is usually too time-consuming. Therefore automatic navigation mesh generators have been created.

An automatic generator of navigation meshes that is very similar to GINT is NEOGEN [11]. NEOGEN makes use of voxelization to create a map of layers in which it can make use of convex decomposition to turn each layer into a single CPG that becomes the navigation mesh. NEOGEN additionally makes use of the rendering pipeline, in this case, the fragment shader, to render a 2D image of each layer's floor plan. Unlike NEOGEN, GINT makes use of a fragment shader and skips the voxelization and convex decomposition to create navigation meshes that can be navigated on with pathfinding algorithms on the GPU instead of the CPU. GINT additionally has the goal of achieving real-time mesh generation speed, which, in this thesis, requires sacrifices to precision and multi-layering of the navigation mesh.

There also exists an extension on NEOGEN called NEOGEN 3D by Gurung [7]. NEOGEN 3D aims to solve some of NEOGEN's existing limitations such as non-manifold meshes; a manifold edge is part of two faces. NEOGEN 3D solves this problem by converting non-manifold meshes into manifold meshes. GIPP does not care about manifold meshes and non-manifold meshes since it is geometry-independent. Both NEOGEN and GIPP have issues with height constraints, it is one part that needs to be set manually and can risk incorrect navigation meshes being generated. NEOGEN 3D solves this issue by adding obstacle edges to limit navigation where the issues arise.

An alternative way of generating a navigation mesh is by using expanding convex shapes to fill the navigation mesh. DEACCON, by Hale, Youngblood, and Dixit [8], is an example of this technique. DEACCON (Decomposition of Environments for the Creation of Convex-region Navigation-meshes) uses expanding quads to create a navigation mesh. DASFV (Dynamic Adaptive Space Filling Volumes), by Hale and Youngblood [9], makes the space-filling navigation mesh generation techniques viable for dynamic game worlds by repairing the navigation mesh in response to changes in the game world. Due to the lack of efficient handling of complex code on the GPU, GIPP does not repair parts of its navigation mesh, instead, it re-renders the entire mesh using the rendering pipeline.

3.4 Path Planning Algorithms

There are several algorithms for path planning, also known as pathfinding, the most common one being A* [15] which finds the fastest path from a starting node to a goal node. This algorithm must be run once for each agent, which becomes more expensive for each agent in the scene. A* by itself is usually too slow for games and thus needs optimized alternatives.

There are attempts to make A* a GP-GPU bound algorithm. Zhou and Zeng [20], introduce GA*. GA* makes use of the GP-GPU via Compute Unified Device Architecture (CUDA) to speed up in comparison to A*. The potential bottlenecks of path planning on the GPU found by Zhou and Zeng are memory for visited tiles and computing the heuristic functions. Overall, GA* can path plan a complete route within a few seconds on a resolution of 10000×10000.

DBA*, implied to be an acronym for Data-Base A*, is a path planning algorithm bu Lee and Lawrence [10], that utilizes pre-computation for paths to reduce search time. DBA* also divides each map into sectors. Combining pre-computation and sector abstractions results in less memory usage but better paths than similar variations of A* such as PRA*.

Another alternative to A* is a vector field. A vector field, for navigational purposes, is a set of points with vectors, often interpreted as directions. Patil, van den Berg, Curtis, Lin, and Manocha [12] make use of vector fields to direct crowd simulations. GINT and HALOS can together create vector fields and potentially be used as an alternative to generate vector fields for crowd simulation.

Line of sight pathfinding algorithms have been done on the GPU previously. Fabias and Kallmann [5], have made a path planning algorithm on the GPU using critical points of obstacle vertices. Compared to CPU pathfinding alternatives, the algorithm is slower. However, the potential of high throughput on the GPU makes it possible to overcome its CPU path planning alternatives. HALOS aims to achieve something similar using Line of sight on a grid system instead, meaning that an increase in the number of vertices should not affect the slowdown of the HALOS significantly.

Path planning on grid-based maps has been done before. Permana et al [13], compare A*, Dijkstra and BFS (breadth first search) on a maze runner game. Dijkstra's algorithm, a greedy approach, iteratively selects the node with the smallest tentative distance from a starting node, updates the distances to its neighbors, and

continues until all nodes are included, ensuring optimality by always selecting the shortest path available at each step. Breadth-first search (BFS), however, explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level in a graph or tree, effectively finding the shortest path from the starting node to all other reachable nodes.

Many grid based path planning algorithms do not find the shortest path due to the angles of movement being restricted, often in four or eight directions. Daniel et al [3], provide a solution for this problem, named Theta*. Theta* finds the shortest path along multiple grids instead of per adjacent grid, thus allowing for any-angle path planning. Any-angle path planning means that the path given can consist of any angle, which can achieve a true shortest path.

3.5 Geometry-Independent Techniques: Shadow Mapping

There are other rendering techniques, with different purposes as well, that share some similarities in their rendering techniques to the render pipeline parts of GINT. One of these is Shadow Mapping, which makes use of the rendering pipeline to create a picture, a texture, of what a light *sees*. Fragments, or pixels, of a game, then translate the geometry data to a pixel on the shadow mapping texture to know whether the pixel is within the shadow of the light or not. In other words, the pixel is in a shadow if the picture taken from the light's point of view is unable to directly see the pixel's position.

GINT's similarity to Shadow Mappings comes from the shared method of making use of the rendering pipeline to create a texture that consists of what it is being viewed, in shadow mapping's case it is from a light, in GINT's case it is from a map view, above oriented north up. Later on positions in the world can be translated to a pixel on the texture, this implies that Shadow Mapping finds out whether it is within a shadow or not and for GINT if the position is a valid point for path planning. GINT was originally created as a way of creating a geometry-independent collision map of a stage using the same rendering pipeline that Shadow Mapping uses.

Figure 3.1: Two pictures, the left is a texture generated by Shadow Mapping. To the right is a GINT generated with the same shader setup. This is the first GINT created, although it was used for collision detection rather than path planning, a different purpose to the implementation of GINT in this thesis makes use of.

3.6 Multiple Sources Single Destination

Multiple Sources Single Destination (MSSD) algorithms are capable of computing several paths from different sources to the same goal position. In other words, it is capable of computing paths for several agents trying to reach the same destination compared to algorithms such as A* which is a Single Source Single Destination (SSSD) algorithm. SSSD algorithms compute only one path between a start node and one goal node. This means that SSSD algorithms must be run once per agent rather than once in total and as a result such an algorithm becomes increasingly expensive for every additional agent. The novel path planning algorithm this thesis implements is MSSD and scales performantly with an increase in the number of agents.

Existing MSSD algorithms, such as the modified Dijkstra's algorithm by Eklund et al [4], is an implemented Dijkstra's algorithm for MSSD to test the problem of vehicle routing during simulated earthquakes, and can offer fast solutions for real-life problems.

Some algorithms are capable of computing MSSD in real-time using machine learning. Xiao et al [19], use Ant Colony Optimisation with Monte Carlo tree search to accomplish a 98% reduction in computation time. Ant Colony Optimization mimics the foraging behavior of ants to iteratively construct solutions by probabilistically selecting paths based on pheromone trails and heuristic information, effectively solving combinatorial optimization problems. Meanwhile, Monte Carlo Tree Search is a heuristic search algorithm that repeatedly samples possible moves in a search tree, guided by random simulations, to find optimal decisions in decision-making processes.

Chapter 4

Method

Although there are two research questions in this thesis, there are three different experiments with different goals to answer these questions. To this end, experimentation is the formal scientific method used in this thesis.... The first experiment measures the precision of the paths generated by HALOS. The second experiment measures the performance in a dynamic world. The third consists of experiments similar to other navigation mesh generation articles to improve the comparability of GIPP with other automatic navigation mesh generation algorithms as well as path planning algorithms. The first two experiments are meant to answer their respective research questions whilst the third experiment is for context and insight due to GIPP being a novel concept.

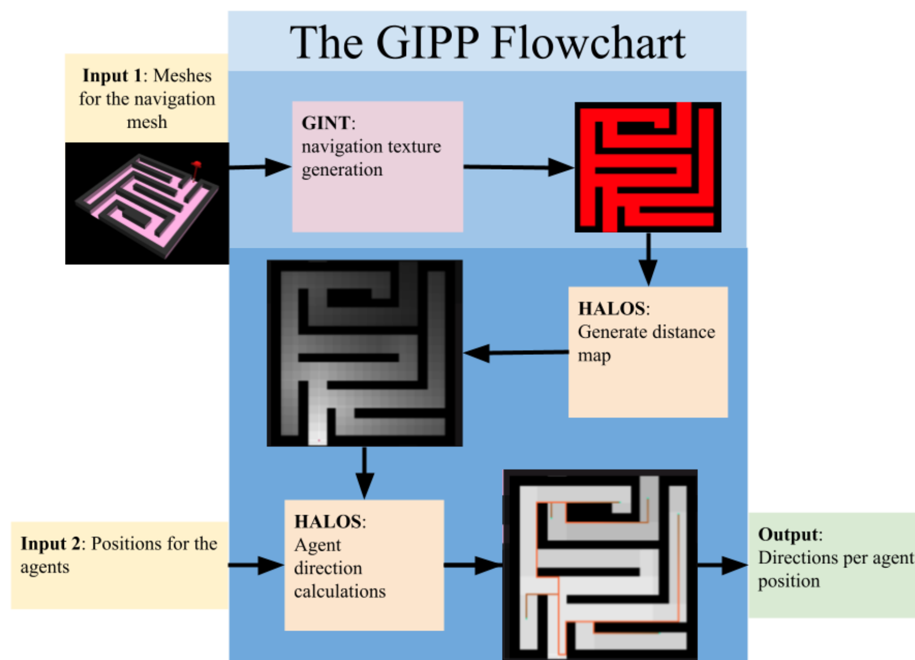

Figure 4.1: A flowchart of GIPP from input to output. There are two inputs for GIPP, the first is the meshes for navigating on. The second input is a list of positions in world space for the agents that will use GIPP for path planning. The output is a list of directions, one direction per agent based on their respective positions.

Algorithm 1 A pseudo-algorithm giving an overview of GIPP

```
//Begin GINT
for GINTToggleOn(Mesh) do
    Render mesh to GINT
end for
//Calculate GINT boundaries based on rendered data
//Finish GINT
//Begin HALOS
while NumberOfBorderTiles > 0 do
    //Explore a new layer and find border tiles
end while
//Get a direction list via a position list
//Finish HALOS
```

4.1 Implementation

GIPP is implemented as two Visual Studio 2022 projects, one of which is a static library and the other is an executable that makes use of the library. The library project contains the GPU shaders for GIPP whilst the executable project runs the experiments. The purpose of having two projects is to make it easier to determine what is GIPP and what is implementation around GIPP. Therefore it becomes apparent what issues are caused by GIPP itself and what issues are independent of GIPP.

4.1.1 ImGui

This project utilizes Dear ImGui [2] to visualize GINT and HALOS as well as enable options for testing ranging from choosing testing scenarios, editing GINT resolution, changing goal position, editing GINT limits, toggling dynamic obstacles, tweaking HALOS variables, toggling time capture for GINT and HALOS, and more.

4.1.2 GINT

GINT is implemented into the library and utilizes DirectX11 to function. This implementation is intentionally made very basic but with some scalability in mind. To render to GINT one simply makes a second draw loop with GINT settings activated. The implemented rendering pipeline for GINT contains a vertex shader and a pixel shader, the DirectX equivalent to the fragment shader. There is also an auxiliary compute shader that automatically calculates the borders to encapsulate all meshes given to the GINT rendering pipeline when the rendering pipeline is finished.

4.1.2.1 Rendering a Mesh to GINT

Initiating the GINT drawing phase requires four function calls. The first, usually, is to clear the GINT map using a function call. After that, or before that, the GINT pipeline is loaded using a function call. After this, each mesh can be rendered by

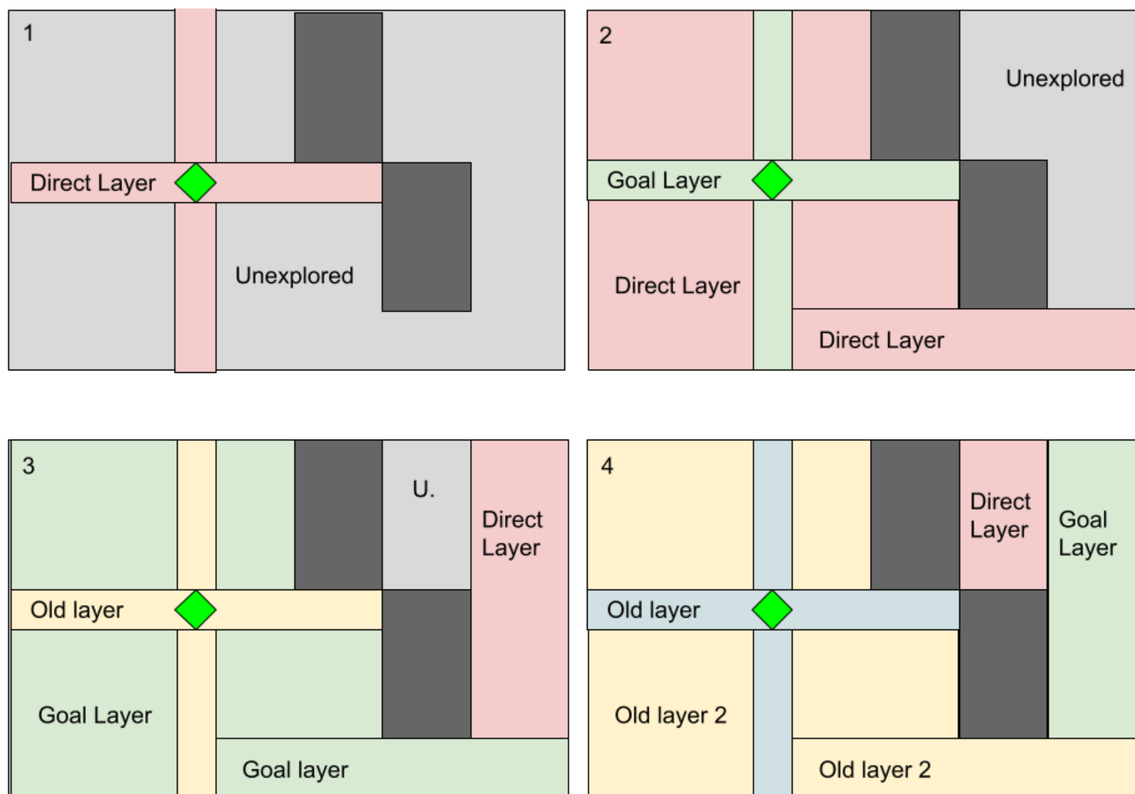

Figure 4.2: Four figures showing each step HALOS makes when generating its layers. A total of four layers are created to explore the entire map. HALOS only explores in four directions: north, south, east, west.

using four different functions. The first three functions set the index buffer, vertex buffer, and world matrix given in their respective arguments. Once given a mesh can be drawn to GINT via a draw function which takes in an additional *GINT type* as its arguments. Tiles on the pixel shader that contain valid height values and normal values according to the settings are given the *GINT type* value. It is possible to set *GINT type* to zero, if so, the entire model is considered an obstacle, regardless of other settings such as height and normals.

The settings are thresholds used to determine walkable and obstacle tiles. These settings include a minimum height threshold, a maximum height threshold, and a y-axis scalar normal threshold. The minimum and maximum height thresholds are limits in world space. Exceeding these limits means a tile becomes an obstacle, regardless of the s*GINT type* value setting. The y-axis scalar normal threshold is between -1 and 1 and represents the y-value of the normal of the tile in world space. A y-axis scalar normal threshold of 1 means that only normals that face straight up are accepted as valid tiles, the other tiles become obstacles. A value of -1 means all angles are accepted as walkable, including upside-down tiles. A y-axis scalar normal threshold of 0 means all tiles that point upwards or perfectly sideways are considered walkable.

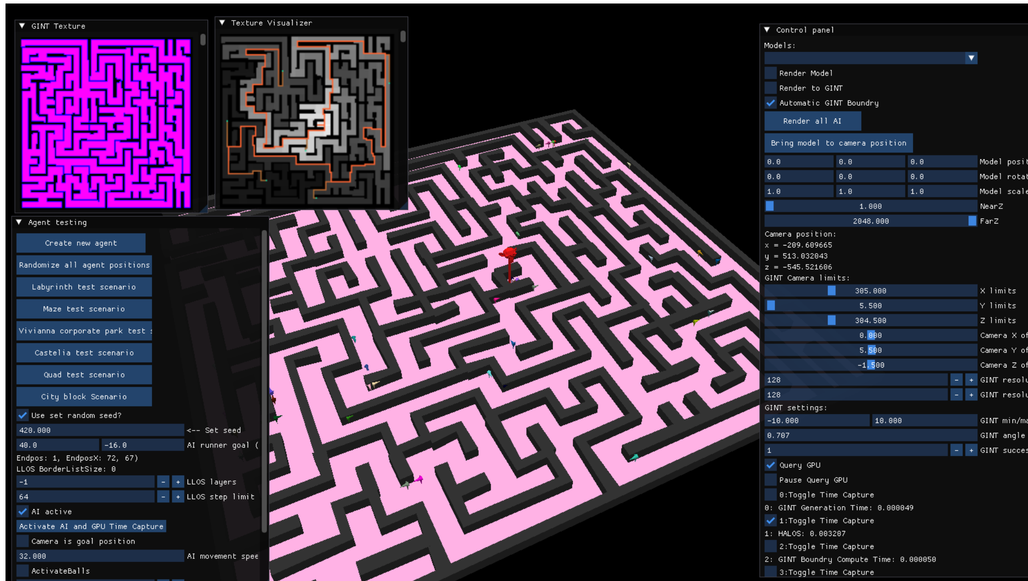

Figure 4.3: This figure shows the ImGui control windows for modifying variables and algorithms during run-time. The pink map in the upper left shows the rendered GINT map. The black and grey map to the right of it visualizes the HALOS algorithm, drawing the generated paths onto it. The other 2 windows named "Agent testing" and "Control panel" allow for modification of run-time variables and scenarios.

4.1.2.2 The GINT Datastructure

GINT is a two-dimensional navigation mesh in the form of uniform quadratic grids. Each grid, or tile for the terminology of this thesis, consists of multiple attributes. These attributes can be whatever one needs, for this implementation, the attributes are type (0 is an obstacle, 1 is walkable), Height (in world-space coordinates), and lastly the angular scalar (-1 means the normal is straight down, 1 means its straight up. Any types between follows the Sinus curve of the unit circle). These three attributes become the information that HALOS needs for its path planning. There is also inbuilt support for types higher than 1, in which case they represent the multiplicative cost of traversing them.

4.1.2.3 The GINT Pseudo-Algorithm

GINT follows a similar path to shadow mapping. A shadow map does not have to include all meshes, only meshes that will cast a shadow are needed in the shadow map. Similarly, GINT only needs meshes that will be the navigable world. The GINT algorithm therefore contains functions that set up and reset the rendering pipeline and functions that render meshes onto GINT. All meshes rendered to GINT are also saved in a list used as input to automatically calculate the boundary, which is the area that GINT is looking at. The boundary can also be adjusted manually, thus the automatic boundary calculation is optional.

Algorithm 2 A pseudo-algorithm giving an overview of the GINT rendering pipeline

SetGINTPipeline($myD3D11DeviceContext$)
for GINTToggleOn($Mesh$) **do**
  SetMeshGPUResources($Mesh$)
  RenderToGINT($Mesh$)
  if $AutomaticGintBoundry$ **then**
    SaveToGINTRenderHistory($Mesh$)
  end if
end for
if $AutomaticGintBoundry$ **then**
  DispatchGINTBondryCalculations
end if
FinishGINTRenderPipeline

Algorithm 3 A pseudo-algorithm of the GINT pixel shader. Input is the per-fragment data provided by the Vertex shader. Global is a constant buffer with set data for all fragments

$Output.height \leftarrow Input.world.y$
$Output.normal \leftarrow Input.worldNormal.y$
if $Output.height > Global.maxHeight||Output.height < Global.minHeight||Output.normal < Global.angleThreshold$ **then**
return $Float4(0, 0, 0)$
else
return $Float4(Input.value, Output.height, Output.normal)$
end if

4.1.3 HALOS

HALOS is implemented as an additional part of the GINT library. Just like GINT, HALOS also makes use of DirectX 11 compute shaders, it also makes use of GINT resources to avoid having to copy the GINT map. HALOS is structured into six compute shaders:

- Reset
- Endpoint
- **Layered Line of Sight**
- **Border Detection**
- Path Planning Directions
- *Visualizer*

The *Reset* and *Endpoint* shaders are only dispatched once each time the endpoint is moved or the GINT map is updated. In the case of this implementation, this is once every frame. The shaders in bold (
textitLayered Line of Sight, Border Detection) are dispatched for each layer of the

line of sight that is generated by the *Border Detection* and *Endpoint* shaders. Once all layers have been dispatched HALOS is ready for directing agents that are on the GINT map towards the end goal using the *Path Planning Directions* shader. One of the six compute shaders is an optional Visualizer that renders the HALOS directions onto a texture for visualization purposes, this is the *Visualizer* shader.

4.1.3.1 The HALOS Datastructure

Though HALOS ultimately outputs a list of directions for each agent it does have its own data structure in the form of a distance map. The HALOS map is structured as a copy of GINT, however, the attributes differ from GINT. Each tile in the HALOS map contains the cost of the tile, same as with GINT, the distance from the tile to the goal node in tile lengths, and lastly the layer. The layer consists of either type -1 if it is unexplored, 0 if it is a goal node and 1 and above is the layer count. Layer 1 means the direct path to layer 0, which is the goal node, layer 2 means a direct path to layer 1, and so on.

4.1.3.2 The HALOS Pseudo-Algorithm

There are two main parts of the HALOS algorithm, how HALOS interacts with GINT and how HALOS explores tiles. Before HALOS begins its exploration, it checks if the goal node is on the GINT map. If the goal node is outside then no border tile, implied to be the goal node, is returned to the CPU from the GPU, in which case HALOS is finished and can either return an error or return nothing for the path planning for its agents.

Algorithm 4 A pseudo-algorithm giving an overview of the HALOS algorithmic loop. The while loop runs until there are no border tiles left, which implies that all tiles have been explored.

HALOSSetShaderResources($myD3D11DeviceContext$)
$NumberOfBorderTiles \leftarrow$ HALOSSetNewEndPoint($GoalWorldPosition$)
while $NumberOfBorderTiles > 0$ **do**
    $NumberOfBorderTiles$HALOSGenerateNewLayer($myD3D11DeviceContext$)
end while
if $VisualizeHALOS$ **then**
    HALOSVisualize
end if
$DirectionList \leftarrow$ HALOSGetPathDirections($AgentWorldPositions$)
HALOSUnsetShaderResources($myD3D11DeviceContext$)

Algorithm 5 A pseudo-algorithm of the HALOS layer exploration compute shader. There are four directions in the implementation: north, south, east, and west.

```
TileData ← GetBorderTileData(ThreadID.x)
if TileData.Layer = Explored then
    Return
end if
Direction ← GetDirection(ThreadID.y)
CurrentTile ← TileData
CurrentPosition
while TileValidExploreMore(CurrentTile) do
    CurrentTile ← ExploreTile(CurrentPosition, Direction)
    if ValidExploration(CurrentTile) = True then
        WriteToTile(CurrentTile)
    else
        Return
    end if
end while
```

4.2 Parallelization and the Sequential Parts of GIPP

The parallel work in the algorithms comes in the form of GPU threads. This means that each dispatch that consists of more than one thread is parallel, assuming the threads are not under constant synchronization. There are mainly two different parallelization methods used in GIPP, the first is the rendering pipeline and the second is the GP-GPU. The rendering pipeline is used by GINT to rasterize one or more meshes into a two-dimensional map. Rasterization, however, leaves the execution to the rendering pipeline of the GPU, which means that certain instructions might not be available on all GPUs. This is accounted for by only making use of the vertex shader and the pixel shader of the DirectX 11 rendering pipeline. The rendering pipeline already has built-in parallelization per vertice and per fragment.

The GP-GPU has two different aspects that are important for parallelization. The first is that there are multiple cores in the GPU, thus allowing for parallel processing. The second is that each core can execute up to 32 threads at once, often called a *Warp*. So long all threads in a warp share the same instructions, a warp can execute all 32 threads at the same time. However, should there be branching instructions, then threads might not share the same instructions, and not all threads in the thread warp can execute at the same time since instructions are no longer shared. To handle this, most compute shaders in HALOS make sure that threads terminate themselves as soon as possible if they need to enter branching instructions.

HALOS has two notable parallelizations. The first is that each border tile contains four threads, each exploring in their directions of up, down, left, and right on the GINT. There is a point of synchronization of the exploration, threads use the Interlocked functions of HLSL when writing the distances to the explored tile. This is relevant since it prevents race conditions where multiple threads of the same layer explore the same tile. Otherwise, race conditions scramble the distances of each tile, causing the HALOS map to no longer represent a heat map toward the goal node.

The second parallelization is that a thread is made per tile on the HALOS map when detecting where the border tiles are.

It is important to know that each dispatch call has an implicit synchronization before the next dispatch call is executed. This means that each dispatch call for the HALOS algorithm is sequential. This also means that the CPU can read the number of border tiles per layer exploration to determine if there are any explorable tiles left. If the number of border tiles costs returns zero, then it means that all explorable tiles, that can reach the goal, have been explored and HALOS can now exit its algorithmic loop and continue to the agent directions part of HALOS.

4.3 Precision

The precision of the path generated by GINT is tested by changing the resolution of GINT while at the same time retaining the same world scenario. To measure the precision, which is interpreted as the ability of any agent to choose the shortest path, the collective time of each agent reaching its destination. A difference in collective time to reach the goal when the resolution differs indicates a difference in precision. An example of an expected outcome is that a lower resolution reduces the precision by having the agents make less accurate path planning as details from walls become lost due to the lowered resolution.

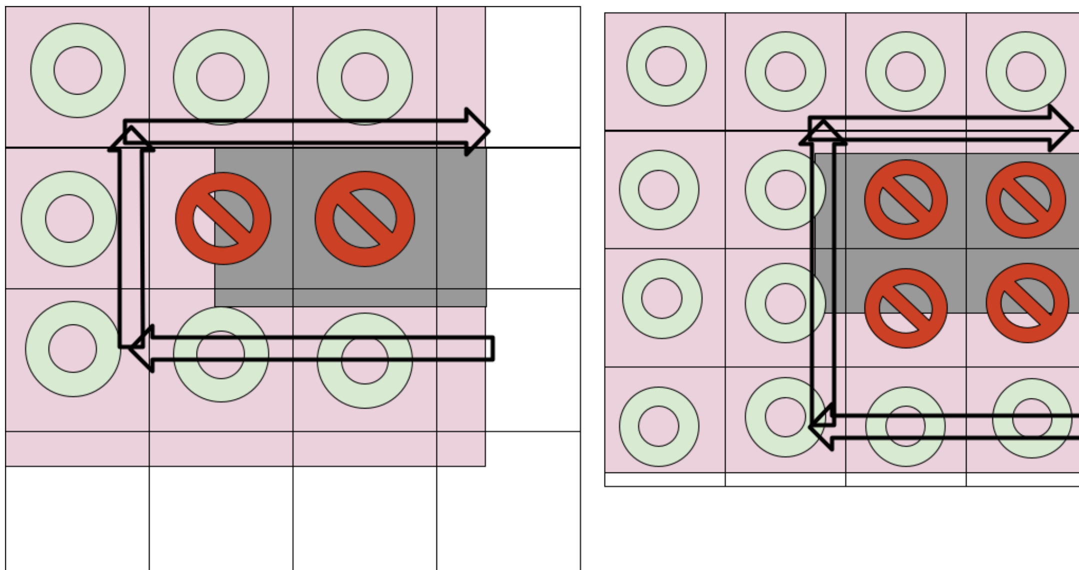

Figure 4.4: Two figures showing how a changing resolution, in this case, shown as pixels with varying size, may affect the route taken by the algorithm. The smaller pixels imply a higher resolution. The green circles define pixels that are valid according to GINT, whilst the red circles with a line define pixels that have an obstacle in the center.

The hypothesis behind this is that the resolution affects the length taken by the agents before a turn. This comes from the fact that each pixel has an in-world length that is based on the dimensions and resolution of the scene. Since the scene is kept the same, only the resolution should affect the pixels. A lower resolution is expected to miss out on details that could otherwise increase or decrease the distance of the path. In 4.4, the pixel size, a result of a changing resolution, may cause the grid map to alter the path taken. There may be cases where a faster path could be taken, however, in the right figure an incorrect path is taken due to the sampling error caused by the pixels. This shows a theoretical imprecision of the rasterizer in GINT when generating the map.

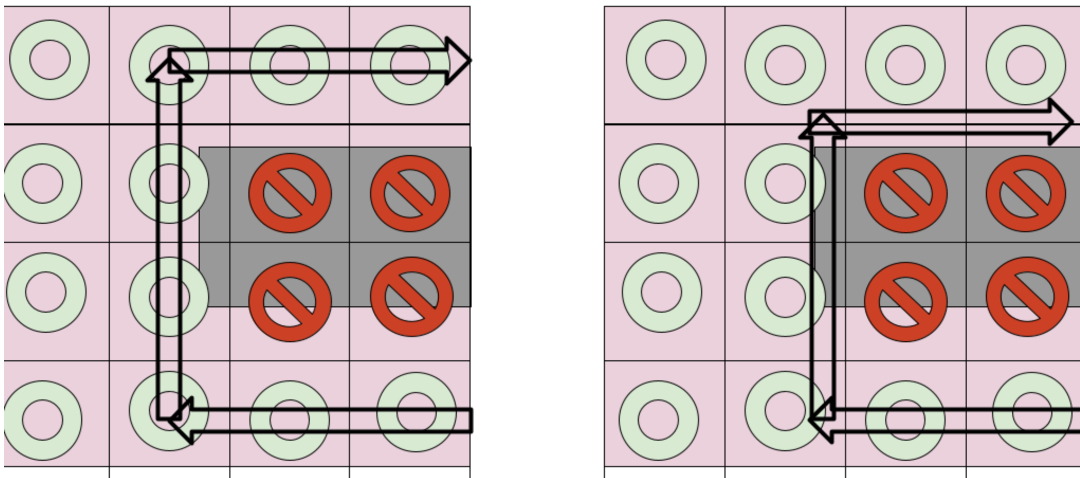

Figure 4.5: Two figures, the left one shows how the path is measured should the entire path be given to the agent. The right figure shows the potential path taken should the agent be given its directions based on its position only; this is how the current implementation of HALOS gives its path to the agents.

Three measurements were taken to find out the precision of the path planned. The first two are the distances collected from each agent to the end goal via the path they're given and via the cost they're given by the HALOS distance map. The third is the collective time it takes for them to walk from their position to the goal in a simulation. The reason for the simulation is to catch cases that pure theory might otherwise miss. The measurements are relevant since the implementation does not move the agents perfectly along its path. These inconsistencies might cause the path to be overall longer, or perhaps shorter in some cases. Both these measurements are needed since HALOS does not give the entire path to the agents at once but instead lets them move in a single direction at a time based on their current position.

Figure 4.6: The subsection of *Castelia City free 3D print model* by *yoshiproductions* chosen for the precision tests. The black parts are obstacles whilst the pink parts are walkable areas.

There is only a single scenario for the precision tests. For ease of reproduction, the game world scenario is a free and publicly available 3d model. In this case, the model is *Castelia City free 3D print model* by *yoshiproductions*. The model is chosen for its mix of wide and small streets. Only a subsection of the city is tested, however. The subsection is the lower part of the city, as seen in 4.6.

4.4 Performance

To measure if GINT and HALOS are viable for real-time multiple scenarios with dynamically moving obstacles, tests with the average, minimum, and maximum per frame time for GINT and HALOS respectively have been taken. The aim is to achieve real-time rendering speeds for the program. In this case, a real-time speed is defined by the program being able to redraw GINT and path plan an entire route with HALOS within 16.7 milliseconds, or one 60th of a second.

The hypothesis behind this is that a dynamic world might cause HALOS to generate a variable layer count, depending on how the world changes, thus causing a variable amount of GP-GPU dispatch calls. This might in turn affect the performance of GINT and HALOS as a whole since the overhead potentially makes up a significant part of HALOS time to finish generating the distance map.

Importantly, the performance tests are done on multiple computer setups. The reason is to better understand if different CPU and GPU setups can affect the performance of GINT and HALOS. This could be useful to find out whether an increase in future computing power affects the real-time viability of GINT and HALOS.

Unlike other path planning studies, this thesis will not make use of the existing maps available at https://movingai.com/ due to there not being any meshes available for the respective grid maps. This means that comparison with other path planning algorithms is more indirect. Another reason for avoiding the same use is that this thesis focuses on the viability of GIPP as a novel technique for both navigation mesh generation and path planning rather than a direct comparison with only existing path planning techniques.

4.5 Auxiliary Data

In addition to the experiments for answering the question, additional data has been taken from the experiments. The purpose is to find out how increases in world complexity, in this case, the number of vertices for the meshes that make up the world, affect the performance to compare how the performance is affected when complexity changes. For this, data from previous navigation mesh generation articles are compared in the discussion 6.

The idea behind the Auxiliary data is to provide data that is easier to compare to other navigation mesh generation and path planning algorithms. This includes past but also future works, should they wish to compare their algorithms to GINT or HALOS. These tests are also made on multiple computer setups but with the reason of providing data that could be potentially more comparable to other studies. Should the computing power differ too much, then useful comparisons could be difficult to make.

Chapter 5

Results and Analysis

The results include visualizations of tables of data taken using the three experiments in 4. The tables can be found in the appendix A. Firstly the renders of the meshes used for the experiments are provided. In addition to the pictures, the number of vertices is also provided. The amount of vertices ranges from 4 vertices up to 1.13 million vertices. The number of agents ranges from 6 to 80 agents.

Six scenes are used for the experiments. These scenarios are Labyrinth, Maze, Vivianna, Castelia, Quad, and City. A picture of each scenario rendered is provided. The visualizations of the scenarios are pink, black, and red. The pink color represents a walkable area for the agents, the black color represents obstacles or out-of-bounds area for the agents and the red color represents the goal position the agents navigate towards.

Four different computer setups are used for the experiments. The computers are given the names: NVIDIA laptop, NVIDIA desktop, AMD laptop, and AMD desktop. Their CPU, RAM, GPU, OS, and compiler versions are all provided in 5.2. The times taken by the experiments are queried GPU time in milliseconds or microseconds, thus the most notable part of each computer is its GPU.

Table 5.1: Number of agents and vertices in each respective scenario

Testing scenario	Number of agents	Number of vertices
Quad	40	4
Maze	6	2856
Labyrinth	50	4448
City	80	122399
Vivianna	30	164262
Castelia	80	1129958

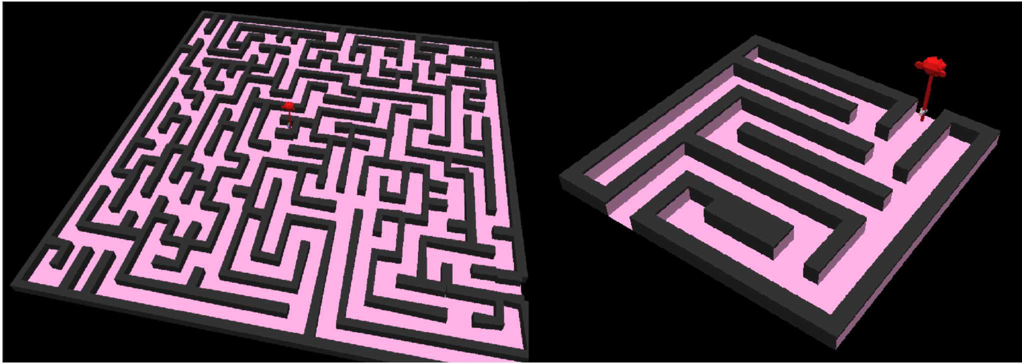

Figure 5.1: This figure shows the Labyrinth (Labyrinth), by *kabeltelevizio*, testing scenario to the left and the Maze (Maze), by *RosieCarter_*, testing scenario to the right.

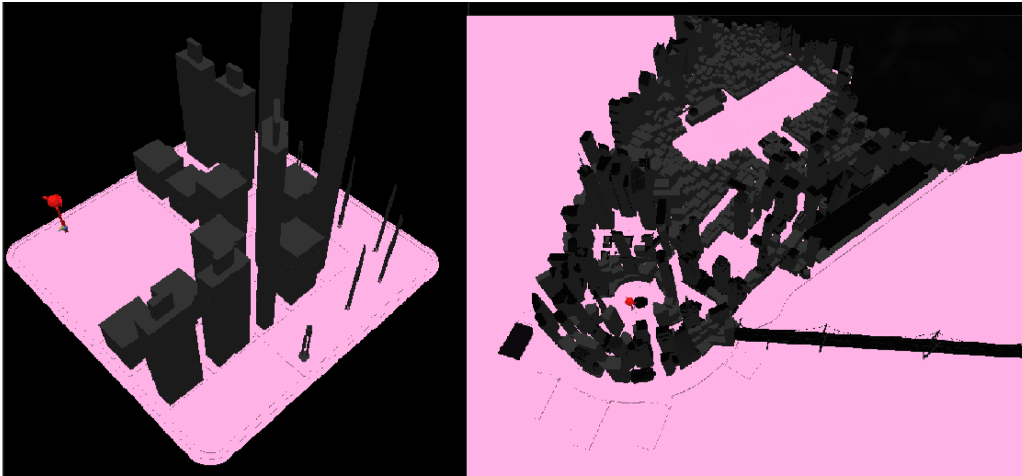

Figure 5.2: This figure shows the Vivianna Corporate Park (Vivianna), by *yoshiproductions*, testing scenario to the left and the Castelia City (Castelia), also by *yoshiproductions*, testing scenario to the right.

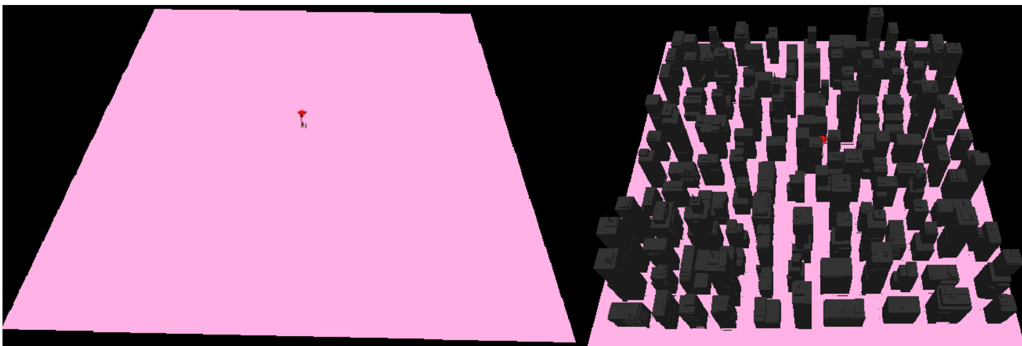

Figure 5.3: This figure shows the Full Quad (Quad), by the thesis writers, testing scenario to the left and the Buildings tall building (City), by Damor3D, testing scenario to the right.

Table 5.2: A table of the hardware specifications of four different computer setups.

Name	CPU	RAM	GPU	OS	Compiler
NVIDIA desktop	Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	16 GB, 2667 MHz	NVIDIA GeForce RTX 2060, 6 GB of V-RAM	Microsoft Windows 10 Home	MSVC v143
NVIDIA laptop	AMD Ryzen 5 7640HS @ 4.3GHz	16 GB, 5600 MHz	NVIDIA GeForce RTX 4060, 8 GB of V-RAM	Microsoft Windows 11 Home	MSVC v143
AMD desktop	AMD Ryzen 7 5800 8-Core Processor @ 3,40 GHz	16 GB, 3200 MHz	AMD Radeon RX 6800 XT, 16 GB of V-RAM	Microsoft Windows 10 Home	MSVC v143
AMD laptop	AMD Ryzen 5900HX with Radeon Graphics @ 3.30 GHz	16 GB, 3200 MHz	ADM Radeon RX 6700M, 10 GB of V-RAM	Microsoft Windows 10 Home	MSVC v143

5.1 Performance Results

Four dynamic performance scenarios using two of the six mesh scenarios, Labyrinth and City are used with the set number of agents in 5.1 but with 16 dynamically moving obstacles. The moving obstacles move in a random direction every other second using a set random seed of 2 (Srand(2)). Each dynamic performance scenario consists of one of the two mesh scenarios but with the GINT resolution of 128 by 128 (128×128) or 512 by 512 (512×512). These are aptly named Labyrinth 128×128, Labyrinth 512×512, City 128×128, and City 512×512 respectively. The GPU time is measured with GPU time queries. The minimum, maximum, and average times are taken of at least 100000 queried samples. GINT and HALOS have separate time queries, both add up to become the total time of GIPP on the GPU. The three figures provided are visualizations of tables A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.11, A.10, and A.9 which are found in the appendix A.

The performance results show that HALOS can maintain an average time of less than 16.7 milliseconds. The average time for the NVIDIA computers ranges from 0.317 milliseconds up to 2.655 milliseconds. The average time for the AMD computers ranges from 1.702 milliseconds to 13.140 milliseconds. NVIDIA computers have lower minimum execution times than AMD computers but also have higher maximum execution times than AMD computers when executing HALOS. The maximum time measured from NVIDIA computers is infrequent and unpredictable, which means not all test runs captured a high maximum time. Both AMD and NVIDIA computers have maximum GPU times higher than 16.7 milliseconds.

The memory usage for GIPP is split into the CPU memory usage and the GPU memory usage. The CPU memory usage is 904 bytes of overhead plus 88 bytes times the number of objects used for GINT every frame (18 objects result in 1584 bytes of memory usage) and 8 bytes per agent (80 agents result in 640 bytes of memory usage). The City scenario results in 3128 bytes of memory usage on the CPU. The GPU memory usage scales with the resolution, the number of objects, the number of agents, and the maximum number of vertices on one single mesh.

The GINT navigation mesh generation times range from 0.003 milliseconds to 6.273 milliseconds. These tests are only taken on the NVIDIA desktop computer. The average times range from 0.057 milliseconds to 0.076 seconds. Again the maximum times measured are infrequent and therefore not sampled on all tests. The automatic GINT boundary computes time averages around 0.18 milliseconds on the Labyrinth testing scenario and 0.45 milliseconds on the City testing scenario.

On the NVIDIA desktop: The combined worst time for GINT and HALOS on GIPP is the City scenario in resolution 512×512 with a sum of 136.831 milliseconds ($98.898 + 31.660 + 6.273$). The sum is the average speed for the City Block scenario with a resolution of 512×512 which is 1.440 milliseconds ($0.889 + 0.069 + 0.455$). The highest sum of average times is the Labyrinth scenario with a resolution of 512×512 with 4.065 milliseconds ($3.805 + 0.076 + 0.184$).

The memory usage on the GPU is, therefore, the sum of 104 bytes of overhead, the resolution in x times the resolution in y times 37 (a resolution of 128×128 results in 606208 bytes V-RAM memory usage), the number of agents times 24 (this is three times the CPU memory usage per agent), The maximum number of vertices on one model times 40 (Castelia has the highest vertice count of 1129958 vertices, resulting

in 45198320 bytes V-RAM usage). The City scenario, in the resolution of 128×128, uses 5104192 bytes of V-RAM, which is in the magnitude of a few megabytes. Note that the maximum number of vertices on one model is considered the mesh itself, this memory can be excluded if the model is not considered unique to GIPP, which results in 608232 bytes of memory usage on the City scenario with a resolution of 128×128. When excluding the mesh data memory, the V-RAM usage for the City scenario with a resolution of 512×512 results in 9701352 bytes, this is 15.95 times more memory than the 128×128 resolution version of the City scenario.

Two scenarios were used to measure the average time for the GINT map to be generated and for the GINT boundary to be updated. For the generation of the GINT map, the most notable difference between static and dynamic was measured at 0.009 milliseconds (9 microseconds), and the least notable difference between static and dynamic was measured at 0.003 milliseconds (3 microseconds). The time for computing the automatic GINT boundary was measured at less than 0.6 milliseconds for the tested scenarios.

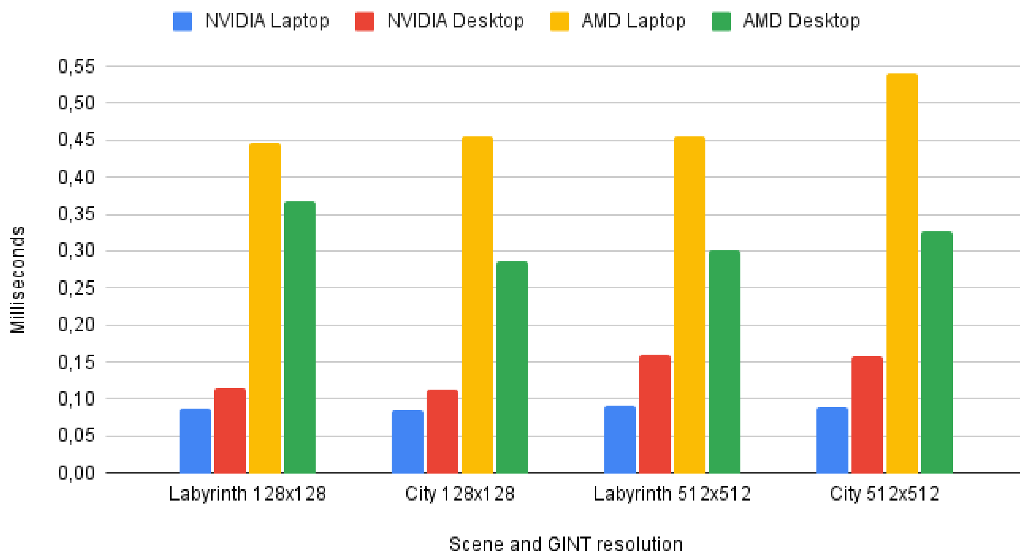

Figure 5.4: The single fastest execution time the HALOS algorithm was executed on, using different hardware specifications and scenarios.

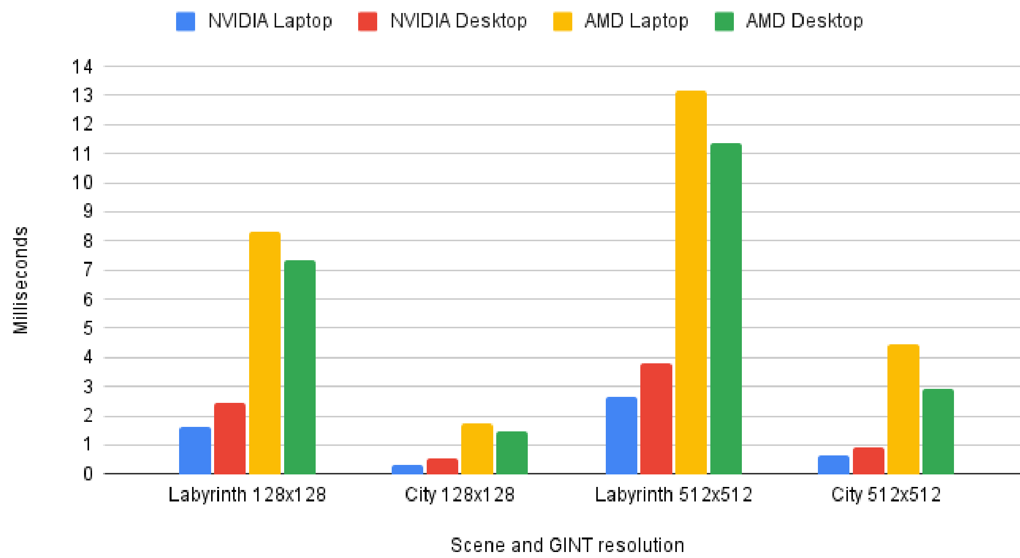

Figure 5.5: The average execution time the HALOS algorithm was executed on, using different hardware specifications and scenarios.

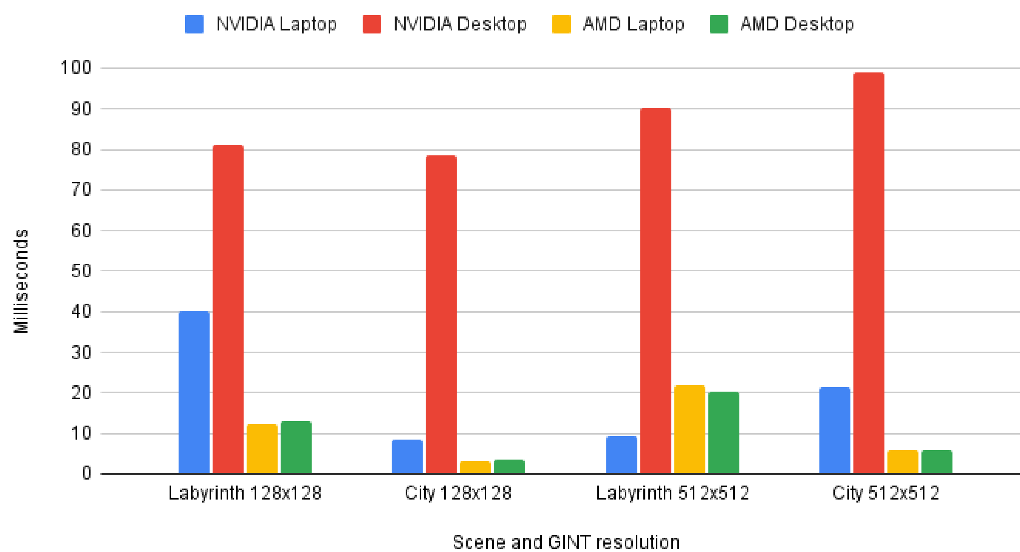

Figure 5.6: The single slowest execution time the HALOS algorithm was executed on, using different hardware specifications and scenarios.

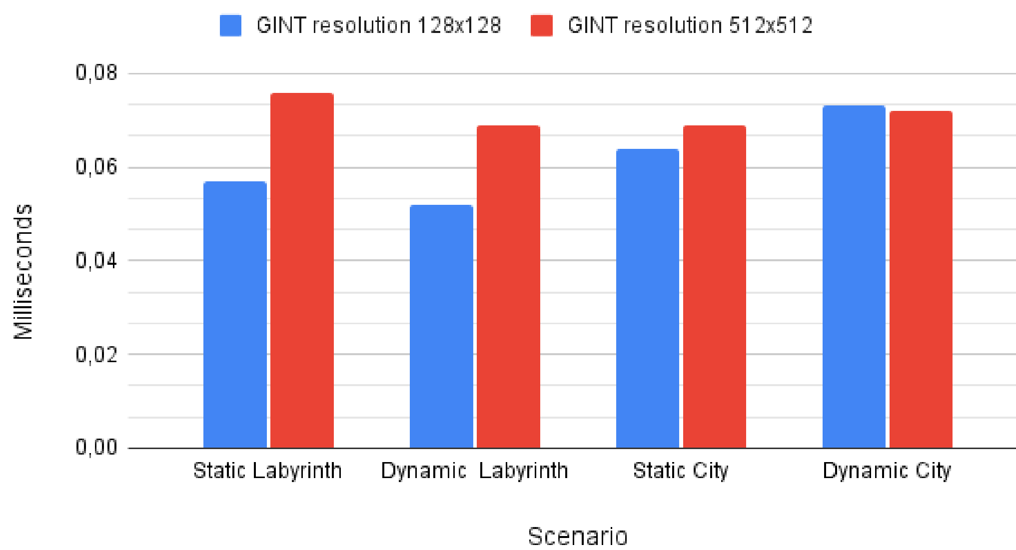

Figure 5.7: The average time it takes to generate both the static and dynamic GINT map of the scenarios Labyrinth and City on the GINT resolutions 128×128 and 512×512. The dynamic element is 16 moving obstacles that agents must navigate around, as such, the navigation mesh must be rendered constantly in real-time to update the location of the obstacles.

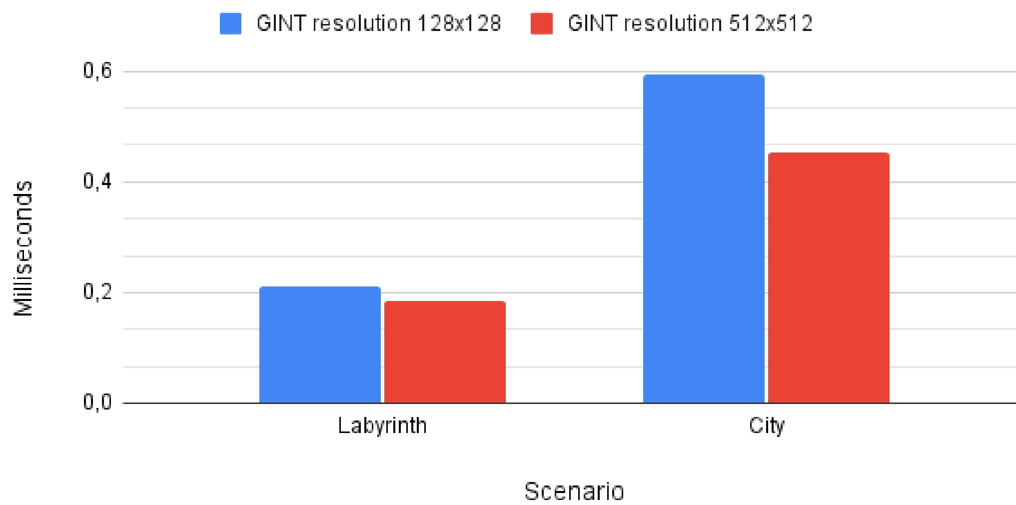

Figure 5.8: The average time it takes to generate both the dynamic GINT boundary of the scenarios Labyrinth and City on the GINT resolutions 128×128 and 512×512. The GINT boundary can only be measured in dynamic scenes as a static scene does not need to update the area at which GINT is looking. The GINT boundary changes when an obstacle moves outside the current GINT boundary.

5.2 Precision Results

The precision results are taken on a single mesh scenario, Castelia, but with varying resolutions. The Castelia scenario was used because of its different widths of streets, which simulate new paths open up as the resolution increases. All resolutions used for the sets are multiples of 32. At the beginning of each experimental test, the tile distance attributes of each tile the agents start on were summed up into the Collective Tile Distance, or Tile attribute. Note that agents that start on non-walkable tiles have a tile distance attribute type of 0. The path from each agent to the goal is also measured and then summed up into the Collective Path Distance, or Path Distance. The distances provided are Manhattan distances. In addition, agents are also simulated to walk in a direction given by their tile every frame until they reach the goal. Should all 80 agents reach the goal, then the collective time to reach the goal is also taken, otherwise, it is assumed to be infinite since not all agents reached the goal. As the resolution increases, the length of each pixel, or tile, on the GINT becomes smaller, therefore an additional figure provides the collective path distance in world length units instead, found in 5.11 The table for the data taken for this test can be found in A.12. There are four visualizations of the table provided below.

The results from the precision experimentation show that there is always inconsistencies between the tile distances generated by HALOS and the path distances provided by HALOS. A tile distance attribute greater than the path distance means that the path generated during the agent direction compute shader generates a shorter path than the HALOS distance map generation did. Additionally, The greatest difference in distances is 2 710 tile units which is a relative difference of 9.47%. In the majority of resolutions, the collective path distance is greater than the collective tile distance.

Not all agents reached the goal node in resolutions of 256×256 and lower. Not all agents reached the goal node in resolutions of 2656×2656 and above. At resolutions of 3520×3520 and higher, no agents reached the goal node. During the simulation of movement towards the goal, the median successful collective time to move to the goal node is 696 seconds. On resolutions of 2048×2048 and higher, when the agents reach the goal node, the collective time is significantly higher than that of the median collective time. These increased times are 920 seconds, 2653 seconds, and 2420 seconds for the resolutions 2048×2048, 2400×2400, and 2592×2592 respectively

The collective path distances have also been converted into world length units, pixel density adjusted length. A world length unit is equal to a translation of 1.0 in world space. The pixel density adjusted length starts at 16744 world length units at a resolution of 32×32 and decreases towards 8046 world length units as the resolution increases. The pixel density adjusted length reached a value within 90% to 110% of 8046 world length units at a resolution of $192x192$. Resolutions 352×352, 640×640, 800×800, and 1024×1024 are exceptions, which are greater than 110% of 8046 world length units.

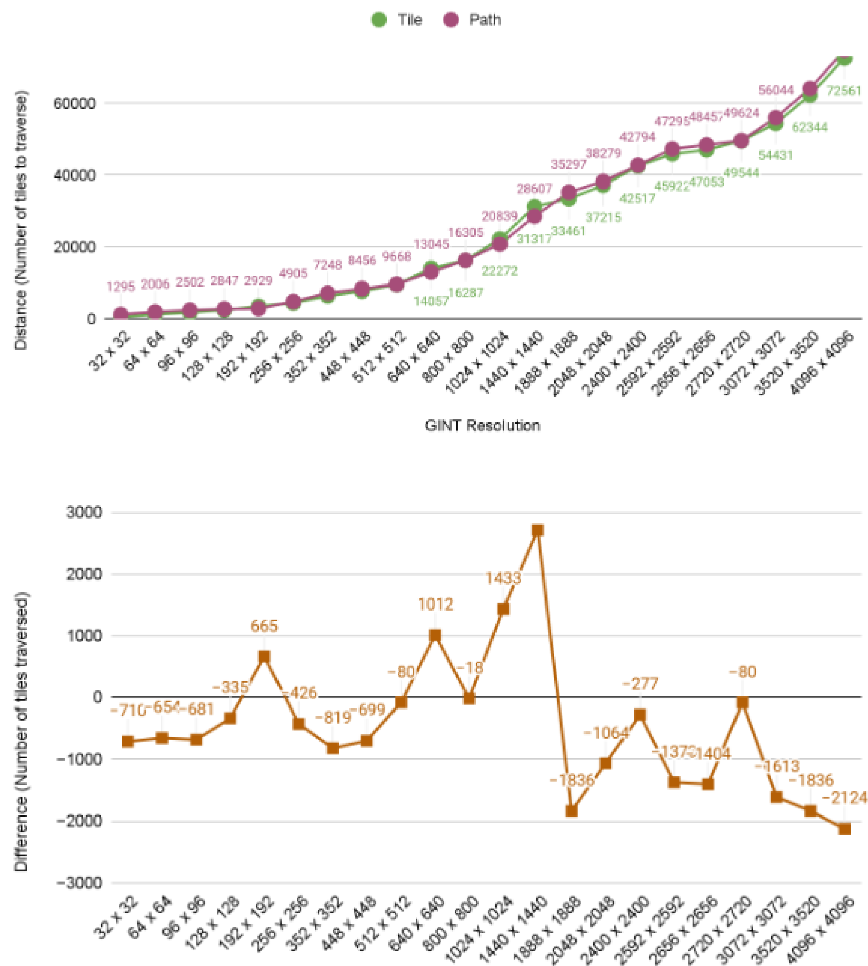

Figure 5.9: The upper line diagram shows the respective Tile attributes and Path distances for different resolutions on the GINT map. The lower line diagram visualizes the difference between the Tile and Path distance from the upper diagram for different resolutions on the GINT map.

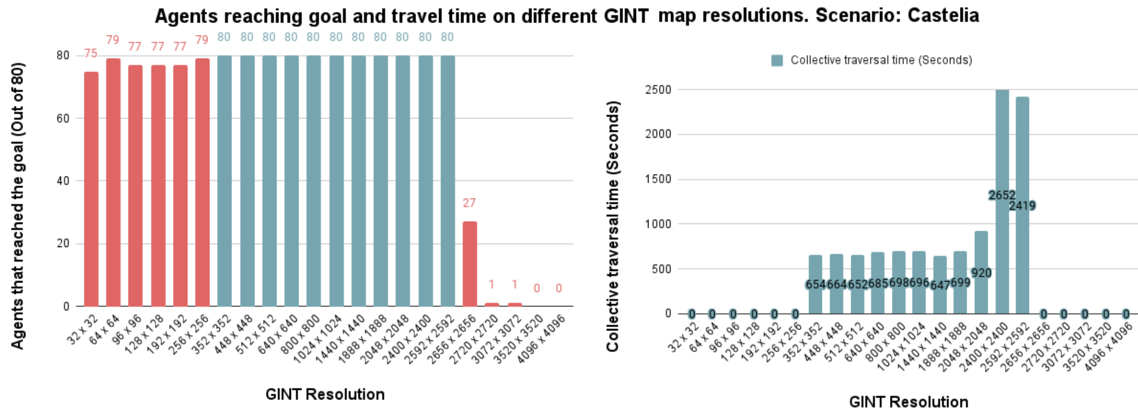

Figure 5.10: The left diagram provides information on how many agents reached the goal, out of the 80 agents for the scenario for various GINT map resolutions. The right diagram shows the collective traversal time of agents reaching the goal in seconds based on the GINT map resolution given. The scenario for both diagrams is Castelia. Note that no final time is measured if at least one agent does not reach the goal.

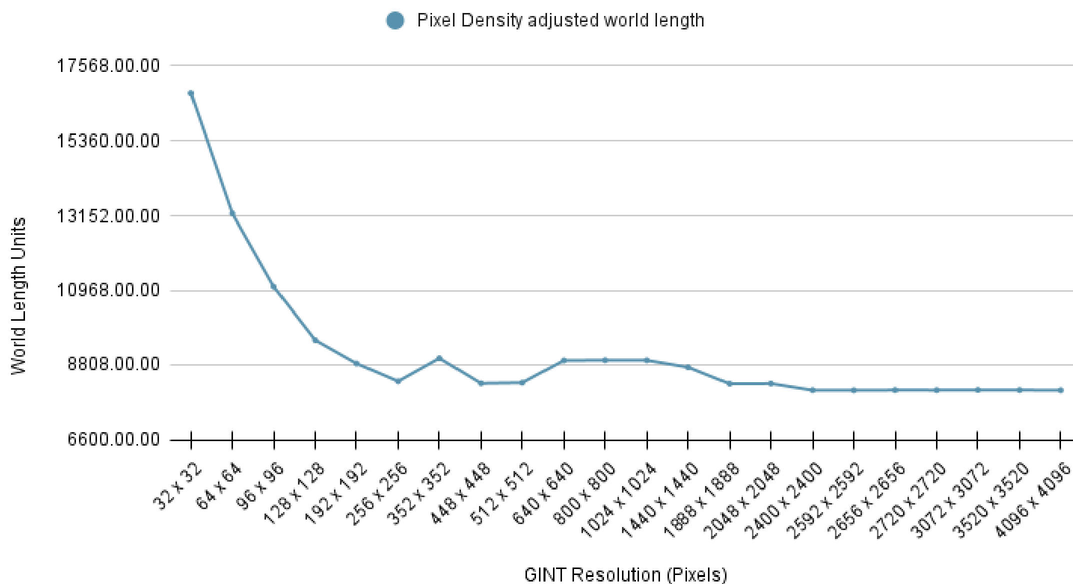

Figure 5.11: This figure shows the collective length of all 80 agents' distance paths when converted from pixels to world-length units. The lines smooth out at 8050 world-length units.

5.3 Auxiliary Data

All six scenarios are run on the NVIDIA desktop computer. The time is measured by querying the GPU from the start of HALOS to the end of HALOS. Note that this includes the execution time of the HALOS visualizer in addition to the rest of the HALOS algorithm. The number of agents and vertices used for each scenario can be found in 5.1. Seven different variations of each scenario are being tested. The difference in each scenario is the GINT resolution, ranging from 32×32 up to 2048×2048. For the visualization of these test, tables A.19,A.18,A.17,A.16,A.15,A.14, and A.13 are being used.

The results of measuring the execution times for the auxiliary tests show that the performance of GIPP increases exponentially with some exceptions. The execution time increases as resolution increases. An increase in the number of layers created for a scenario significantly increases the average execution time of HALOS.

An additional test measuring the performance of the number of agents on the GPU is also performed. The scenario used is the Quad with varying numbers of agents, ranging from no agents up to 2048 agents. Quad is used since it has only four vertices and no obstacles, thus minimizing the execution time differences that might have been caused by variables other than the number of agents. Note that the GPU time includes some synchronization with the CPU, which can warp the time taken for the results. The data used for the visualization of these results can be found in table A.24.

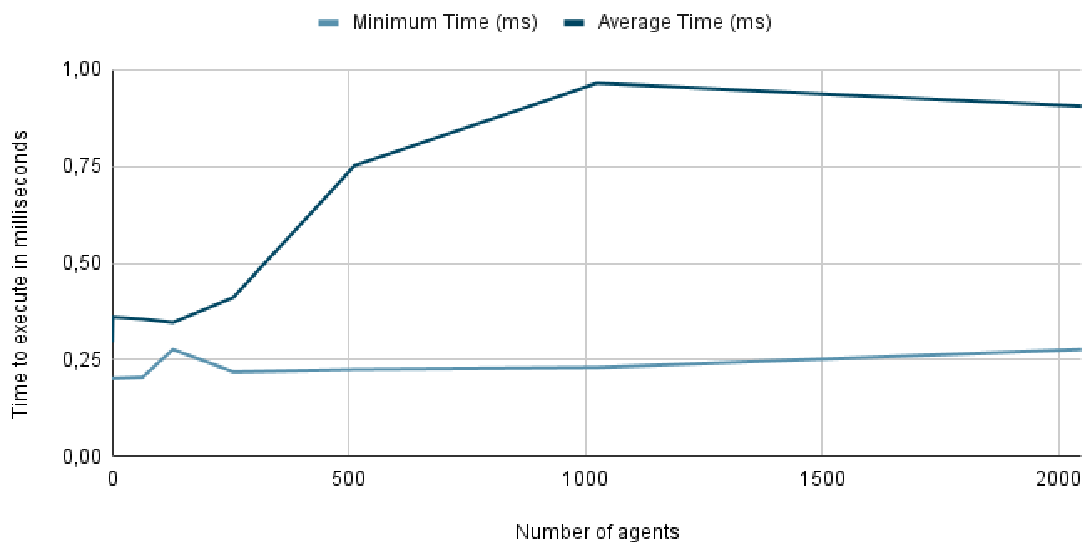

Figure 5.12: The average time to execute HALOS depends on the number of agents given on the Full Quad scenario. The time for having 0 agents is 0.296 milliseconds, this can be considered the overhead of HALOS generating the path planning map. The time for finishing the GPU time query is taken when the CPU receives all the agent directions. Note that this implies synchronization and memory transfers between the CPU and GPU.

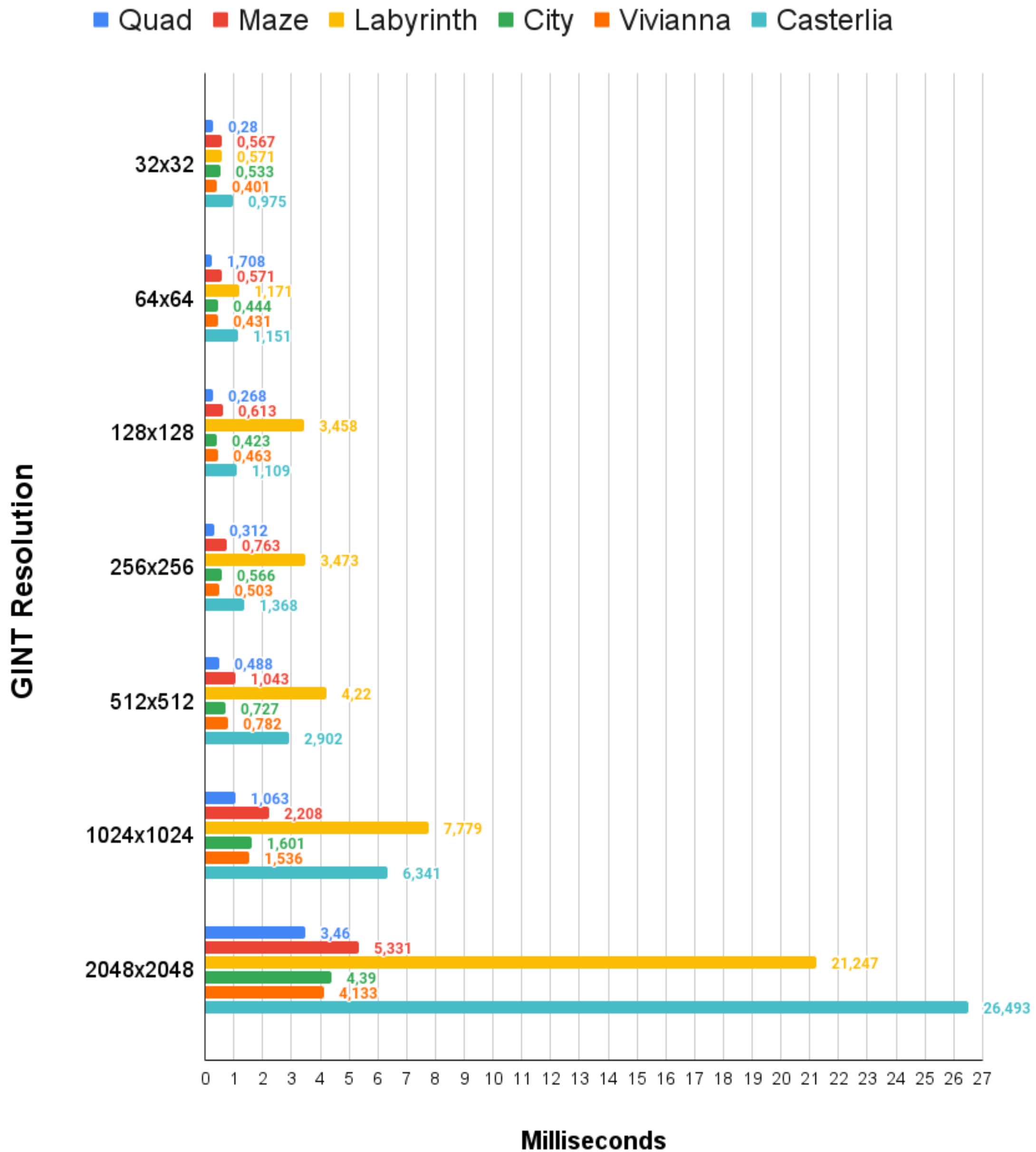

Figure 5.13: The average time to execute HALOS depends on the resolution of GINT for each scenario. HALOS includes the path planning map generation and returning the directions to each agent.

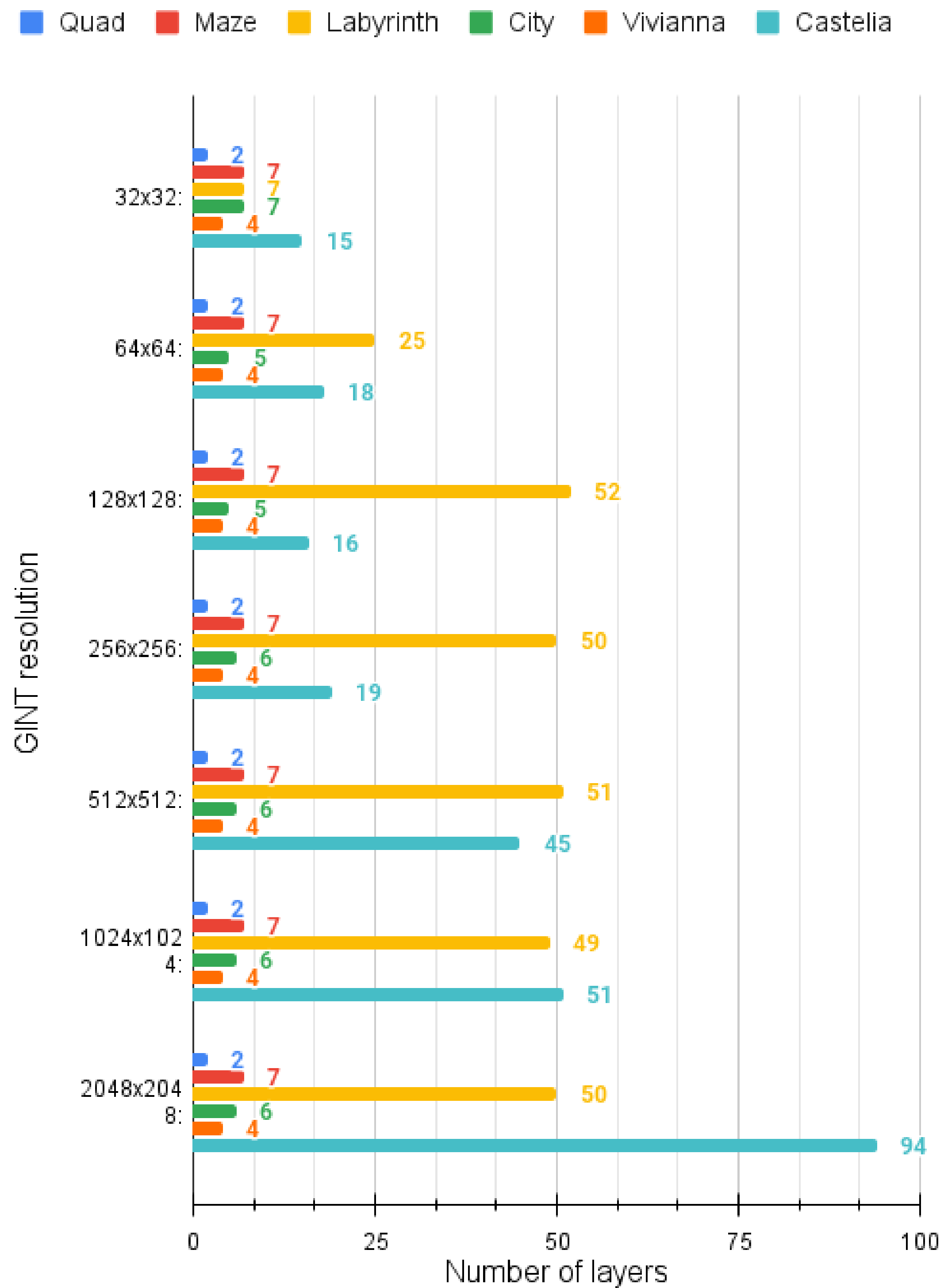

Figure 5.14: The amount of line-of-sight layers HALOS has to generate on different resolutions for each scenario. HALOS generates layers until there are no unexplored tiles bordering explored tiles left.

Chapter 6

Discussion

6.1 Performance

What the results indicate for the first research question:
RQ1 *How do dynamic environments affect the* ***performance****, in terms of execution time and memory, and what is the real-time viability of GIPP?*
Comparing the average execution times of HALOS on the NVIDIA desktop shows no significant differences in performance on the City scenario. However there is a decrease in the average execution time of HALOS on the dynamic labyrinth compared to the non-dynamic scenario. The reason for the decrease in frame time is that the obstacles block paths in the labyrinth, causing a decrease in layers created, thus reducing the execution time of HALOS. Each layer can be considered equivalent to a region in more traditional navigation mesh structures. Having fewer regions is usually desirable since regions cause computational overhead. Creating a new layer requires overhead in the form of two dispatch calls and two memory copies between the CPU and the GPU. The scenarios and resolutions had no noticeable difference in performance for the GINT generation.

Comparing the number of vertices, found in table 5.1, that the City scenario and Labyrinth Scenario contain with the time measurements of generating the GINT map in figure 5.7 suggests that the main bottleneck for scalability of GINT might be memory usage. However, more data on GINT generation is needed to back up this statement. For HALOS, the auxiliary figures 2.2 and 2.1 suggest that both the resolution of the grid map (GINT resolution) and the layer count for generating the HALOS map are bottlenecks for scalability.

6.1.1 Memory

Memory is more ambiguous on GIPP, since it uses the rendering pipeline. Any overhead from the rendering pipeline could be considered part of GIPP's memory usage. However many parts of the rendering pipeline are reused by other parts of the program, independent from GIPP, making it difficult to determine what is GIPP memory. Consider meshes, since GIPP reuses the same meshes used for rendering to the window, one could argue that GIPP has no memory overhead when it comes to meshes since there are no low polygon count meshes or other alternatives needed to make GIPP efficient. GPU resources that keep track of world matrices can also be independently used from GIPP and are therefore considered independent too.

The results in figures 5.9, 5.10, and 5.11 suggest that a resolution of 512×512

provides a good balance of precision without too many details being lost from the rasterization process. The expected memory usage that a resolution of has 512×512 on the CPU RAM is around 3 KB, courtesy of 5.1. The CPU memory only exceeds 10 KB when there are hundreds of objects and agents. The memory usage on the CPU is within the kilobytes compared to the megabytes of RAM used by navigation mesh generators from previous studies. However, the V-RAM, the GPU memory usage, is around 10 MB of V-RAM usage, for the same resolution of 512×512. This is bad but manageable, however, V-RAM usage scales poorly with larger resolutions in comparison to CPU RAM usage.

6.1.2 Comparison

Comparing the results of different grid-based path planning gives a better view of GIPP. The DBA* algorithm [10], as mentioned in 3, uses around 50 to 250 KB of RAM for similar map sizes to a resolution of 512×512. GIPP ranges in kilobytes of RAM, but in megabytes of V-RAM, this is more memory usage than DBA*, however, GIPP skips the pre-computation, which takes anywhere from 0.1 to 10 seconds for DBA*, for the ability to generate the grid map in real-time. The path planning time is considerably slower on GIPP than DBA*; DBA* can navigate maps within microseconds while GIPP takes upwards of tens of milliseconds for complex maps. Here the performance advantage of pre-computation is clear, however, the inability to handle dynamical grid maps might be a weakness of DBA* compared to GIPP.

The results show that GIPP can re-render an entire navigation mesh in under 100 microseconds using GINT. This is considerably faster than DASFV which takes over 100 milliseconds. However, note that the experimentation for DASFV was taken on a 2.13 GHz CPU with 2 GB RAM, which is slower than the NVIDIA desktop's 2.9 GHz, which can boost its clock speed to up to 4.1 GHz. The RAM is not comparable for two reasons, we do not know the frequency of the RAM, and the memory used for DASFV and GIPP does not exceed 2 GB. In either case, the NVIDIA desktop is around two times faster, which does not compensate for the navigation generation being 1000 times faster. However, the CPU comparison is not useful information since the generation of the GIPP navigation mesh is primarily calculated on the GPU rendering pipeline, which does not have a CPU counterpart. Regardless, GIPP is multiple magnitudes faster at dynamic navigation mesh generation than other comparable automatic navigation mesh generation algorithms.

GIPP has issues with tunnels and obstacles far above walkable areas, such as a roof. This results in pathways, like an archway in the Castelia City scenario, being considered an obstacle when it would otherwise be considered a path. NEOGEN, especially its expanded version, should be able to handle this case better than GIPP. However, it is arguable whether or not NEOGEN can generate a navigation mesh for the over one million vertices of the Castelia City scenario within a reasonable time. This is where GIPP shines in comparison; GIPP can handle large and complex meshes at a real-time speed, at the cost of losing out on details and less precise paths.

6.1.3 Issues with the implementation

In this implementation of GIPP, the navigation mesh is recreated every frame. The path planning map is also recreated every frame. Recreating the navigation mesh and path planning map is only necessary should the game world change or the goal node move to another tile. Therefore the performance results can be interpreted as the worst case scenarios of GIPP. Considering the results as the worst-case scenarios, the real-time performance answers the research question about the performance of GIPP better still. A game, a probable user of GIPP, needs time for more than just navigation. In this case, the fact that GIPP requires multiple milliseconds to generate a navigation mesh and path planning map might be too slow, considering there needs to be time left for the other parts of a game. GIPP can achieve real-time speed, however, it is still too slow for most games, which do not have enough time left over to allocate milliseconds every frame for GIPP.

The implementation also has issues with the directions given to the agents. The directions only move in four directions (north, south, west, east), with no diagonal movement. This implementation is a limitation caused by the fact that the current implementation does not discern where in a tile the agents are standing and thus avoids moving into corners instead of around them. Allowing the GPU to get better access to where a tile the agents are in would increase the ability to generate a shorter path by allowing more than four directions of movement.

6.2 Precision

What the results indicate for the second research question:
RQ2 *How does the resolution of GIPP affect the* ***precision*** *of finding the shortest path?*
Comparing GIPP's precision to other studies shows that its coverage, that is how accurately it covers the geometry, is inaccurate and causes paths through obstacle geometry, as the results from high-resolution shows. The results also show that the path distances and tile distance attributes differ on all tested resolutions, which indicates that there are always inaccuracies with GIPP's path planning.

The precision results also indicate issues with the implementation itself. On the greater resolutions, the results show that GIPP can generate a complete path toward the goal node, however, the simulation of the agents is not able to follow the path due to the stepping range of each agent eventually becoming larger than the size of each tile on the navigation mesh. This ends up causing the agents to step over tiles that might otherwise provide important details on their path, causing them to get stuck. A resolution of 512×512 is optimal for precision when accounting for the precision results on the Castelia scenario. The resolution might yield the best results since the speed of all agents remains constant over the various resolutions. The speed is the stepping range per frame based on the frame time. If the stepping range affects the precision of GIPP, then the precision results are faulty and do not represent the actual precision of GIPP, thus diminishing the reliability of the precision experiments.

6.3 GIPP Usability

The use cases of GIPP are therefore best for software that requires generating a navigation mesh within real-time. However, GIPP is not suited for meshes with multiple floors. It is also unsuitable for tunnels. These limitations result in GIPP mostly only being viable for games that contain a single floor. Although GIPP can handle multiple agents that share the same goal node. Therefore games that are suitable for GIPP are top-down games. Examples of games that might make good use of GIPP include Starcraft 2, Diablo IV, and Fall Guys. GIPP's fast navigation generation but slow path planning means that GIPP is most suited for use with testing maps during their development. This means that GIPP might be viable for dynamic game worlds but is outcompeted by pre-computational path planning algorithms on static game worlds.

Although GIPP is still vertex-based, its performance bottleneck is not the number of vertices each mesh contains. Instead, the performance bottleneck is the number of layers created for the HALOS algorithm. The current implementation of GIPP is not particularly impressive with its path planning, however, it is a clear stepping stone towards geometry-independent path planning. GIPP also extends the current research on MSSD GPU path planning algorithms with a grid-based path planning algorithm.

6.3.1 Advantages and Disadvantages of GIPP

The main advantage of GIPP is its geometry-independent mesh generation, meaning that any model is usable for navigation mesh generation, regardless of whether it is adapted to being a navigable area. This means that vertices don't have to follow a certain pattern, such as with height maps, or have wall shapes that can be easily made into one or more convex shapes. The consequence this has on game development is that a game world can quickly be set up and tested without having to spend time and effort designing it for the limitations of other navigation mesh generation techniques, which are often limited obstacles with a set of convex shapes.

A downside of using the rendering pipeline for GINT generation is that it is limited by the maximum texture size of 16384×16384. Considering the hardware of 2024, 16384×16384 is a mostly pointless resolution for GIPP since it is too large for real-time path planning. Another downside of the rendering pipeline is that GIPP suffers from a sampling issue. The single-point sampling rate used for this implementation of GIPP results in bad coverage. This negatively affects the precision of GIPP. Increasing the sampling rate will require a more complex pixel shader which slows down the navigation mesh generation of GIPP. An increased sampling also negatively affects the performance which further slows down the navigation mesh generation time. However, navigation mesh generation is a tiny part of the overall execution time of GIPP. Increasing the sampling rate might therefore not be an issue.

HALOS, the path planning algorithm of GIPP, is well suited for having multiple agents path planning onto a single goal node. The results show that HALOS can handle 1000 agents at 0.965 milliseconds compared to 0.203 milliseconds for a single agent. The reason the time per agent increases is due to there being synchronization between the CPU and the GPU. Implementing the agents' movement on the GPU

instead of the CPU would skip this step and save overhead which can theoretically reduce the path planning time below 0.2 milliseconds regardless of the quantity of agents.

Chapter 7

Conclusions and Future Work

7.1 The Performance of GIPP

GIPP achieved the real-time frame time of an average execution time of 16.7 milliseconds. There are CPU to GPU bottlenecks in the implementation which slows down the GPU considerably due to synchronization between the CPU and the GPU. GIPP is not always able to maintain its execution time under 16.7 milliseconds, which makes GIPP unsuitable for real-time software that needs consistent performance. The performance of GIPP is most negatively affected by the number of layers created. The number of agents and vertices scales well in high numbers for GIPP. The most performant part of GIPP is its navigation mesh generation; the least is its path planning.

When connecting the results to **RQ1**, GIPP is slow but capable of handling navigation mesh generation and path planning in a few milliseconds. GIPP might still be viable for dynamic worlds but it is apparent that GIPP is not a viable contender when it comes to path planning on static worlds. Overall GIPP is a step forward, performance-wise, for generating navigation meshes for dynamic game worlds.

7.2 The Precision of GIPP

The results of the experiments show inconsistencies in how the shortest path is generated on all resolutions on GIPP. It is not apparent if inconsistencies are mainly caused by the implementation of the simulation or the navigation mesh being a rasterized grid map. Connecting the experiments to **RQ2** shows that increasing the resolution of GIPP results in a shorter path. However, the inconsistency of the results for the precision experiments suggests that too much information is lost either due to GIPP's navigation mesh or the simulation itself. In any case, the ability of GIPP to generate the shortest path possible, considered the ideal precision, is lacking and therefore answers the research question on the performance of GIPP negatively.

7.3 Future Work

GIPP shows promising future work since the current implementation has fixable flaws and a large number of potential improvements to both the complexity of meshes GIPP can handle and the performance and precision of GIPP. An improvement to be made for GIPP's precision is the sampling rate for each pixel on the GINT.

Although increasing the sampling rate comes at the cost of performance, which might cause it to lose its run-time performance, should the sampling rate negatively affect performance. For static maps, generating multiple GINTs with variable resolution on the same mesh might provide a performance boost where there are no obstacles but still retain the detail for navigating obstacle-rich areas of GINT. Keeping track of multiple GINTs of varying details increases the complexity of the code for GIPP. Complexity is a considerable weakness of the GPU which might make it difficult to implement such a method.

7.3.1 Optimization

GINT's bottleneck is memory usage. Optimizing the memory layout of GINT will result in both less memory used and potentially more performance timewise since the GPU usually speeds up with less memory usage due to the memory bandwidth of the GPU. HALOS's biggest bottleneck is algorithmic, in other words finding ways to reduce the amount of layers created will result in a speed-up. One solution to reduce the number of layers is to create layers until all agents are positioned in a layer skipping unnecessary unexplored tiles.

7.3.2 Multiple GINTs

Having multiple GINTs opens up the possibility of having multiple floors. Multiple floors allow GIPP to handle more complex meshes, however, the problem of defining what a floor is means that there are challenges with how GIPP should determine when a new floor should be made. Automating the determination of floors is a challenge itself, independently from GIPP. The HALOS path planning algorithm can handle multiple GINTs so long as there are attributes in tiles that can link to other GINTs. Both GINT and HALOS need to be Expanded upon for GIPP to handle multiple floors.

7.3.3 MSMD

HALOS has innate support for Multiple Source Multiple Destination (MSMD) path planning since the border and layer part of the HALOS algorithm creates a list of goal nodes by practicality. Implementing the possibility of creating multiple goal nodes will result in each source finding the shortest path to either of the goal nodes. Reversing the cost of each goal node will result in agents avoiding all goal nodes, which might have potential use for finding paths that avoid harmful nodes. Expanding HALOS to support MSMD is therefore a simple task since only the goal node compute shader has to be remade to handle multiple goal nodes as input.

7.3.4 Other Rendering Methods

GIPP can theoretically handle other rendering methods than vertice rasterization. GIPP has the potential to be expanded to generate GINTs of ray-marching results which opens up the possibilities for navigating Fractals from a 2D space, the practical use for this is unknown, however. GINT, the navigation mesh generation, part of

GIPP has potentially other uses than just representations for path planning. One could rotate the camera GIPP uses to look at a mesh from the direction of velocity, then the GINT map can be reinterpreted as a map for air resistance instead. Such a solution can bring a real-time, but low-precision, solution to having air resistance in games for example.

7.3.5 Reducing Overhead

CPU-to-GPU communication is a significant bottleneck for achieving path planning performance in under a millisecond for GIPP. Restructuring HALOS to make dispatch calls from within the GPU instead of the CPU, thus avoiding the CPU, significantly reduces the performance overhead of GIPP. Implementing such a change would require a complete rewrite of HALOS in addition to potential synchronization problems. Simulating the movement of agents via the GPU instead of the CPU is another part of overhead that can be reduced.

7.3.6 Overall Specialization Route

GIPP is most likely a specialized tool for handling large quantities of simple path planning tasks on complex dynamic worlds. Embracing this specialization is likely the best route for further studies of GIPP since GIPP's usage of the GPU, a unique part of GIPP, is already specialized for simple but many parallel instructions. This means that GIPP is not recommended for further studies on tasks that need complex path planning, have few parallel parts, and utilize highly detailed static worlds.

References

[1] N. Amato and Y. Wu, "A randomized roadmap method for path and manipulation planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, 1996, pp. 113–120 vol.1.

[2] O. Cornut, "Dear imgui," *URL: https://github. com/ocornut/imgui*, vol. 5, no. 8, 2024.

[3] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, p. 533–579, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1613/jair.2994

[4] P. Eklund, S. Kirkby, and S. Pollitt, "A dynamic multi-source dijkstra's algorithm for vehicle routing," in *1996 Australian New Zealand Conference on Intelligent Information Systems. Proceedings. ANZIIS 96*, 1996, pp. 329–333.

[5] R. Farias and M. Kallmann, "Optimal path maps on the gpu," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 9, pp. 2863–2874, 2020.

[6] F. M. García, M. Kapadia, and N. I. Badler, "Gpu-based dynamic search on adaptive resolution grids," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 1631–1638.

[7] S. Gurung, "Dynamic updates of navigation meshes," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:208098562

[8] D. H. Hale, G. M. Youngblood, and P. N. Dixit, "Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds," in *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'08. AAAI Press, 2008, p. 173–178.

[9] D. H. Hale and G. M. Youngblood, "Dynamic updating of navigation meshes in response to changes in a game world," in *The Florida AI Research Society*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:9897104

[10] W. Lee and R. Lawrence, "Fast grid-based path finding for video games," in *Canadian Conference on AI*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:7061540

[11] R. Oliva and N. Pelechano, "Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments," *Computers & Graphics*, vol. 37, no. 5, pp. 403–412, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849313000435

[12] S. Patil, J. van den Berg, S. Curtis, M. C. Lin, and D. Manocha, "Directing crowd simulations using navigation fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 2, pp. 244–254, 2011.

[13] S. H. Permana, K. Y. Bintoro, B. Arifitama, A. Syahputra *et al.*, "Comparative analysis of pathfinding algorithms a*, dijkstra, and bfs on maze runner game," *IJISTECH (International J. Inf. Syst. Technol*, vol. 1, no. 2, p. 1, 2018.

[14] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[15] E. Skublewska-Paszkowska, "A* pathfinding algorithm modification for a 3d engine," 2019.

[16] G. Snook, "Simplified 3d movement and pathfinding using navigation meshes," in *Game Programming Gems*, M. DeLoura, Ed. Charles River Media, 2000, pp. 288–304.

[17] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[18] W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts, "A comparative study of navigation meshes," in *Proceedings of the 9th International Conference on Motion in Games*, ser. MIG '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–100. [Online]. Available: https://doi.org/10.1145/2994258.2994262

[19] Z. Xiao, C. Liu, S. Luo, K. Huang, H. Gao, X. Xu, and X. Wang, "A collaborative and dynamic multi-source single-destination navigation algorithm for smart cities," *Sustainable Energy Technologies and Assessments*, vol. 56, p. 103032, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213138823000243

[20] Y. Zhou and J. Zeng, "Massively parallel a* search on a gpu," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, Feb. 2015. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/9367

Appendix A

Supplemental Information

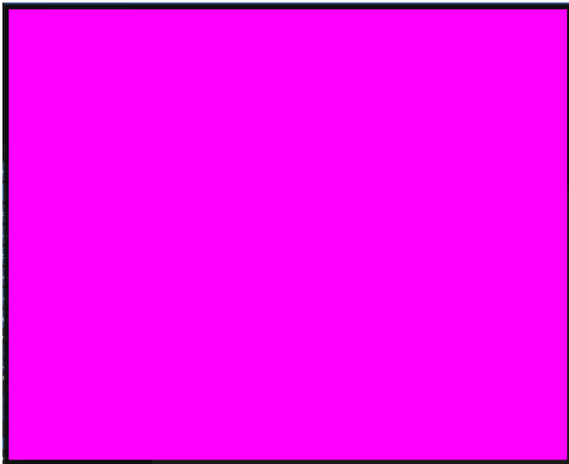

Figure A.1: This figure shows the GINT map of the quad testing scenario at 2048×2048 GINT resolution.

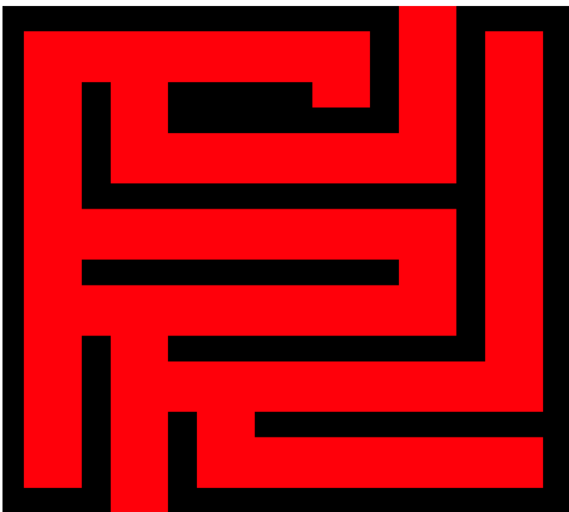

Figure A.2: This figure shows the GINT map of the maze testing scenario at 2048×2048 GINT resolution.

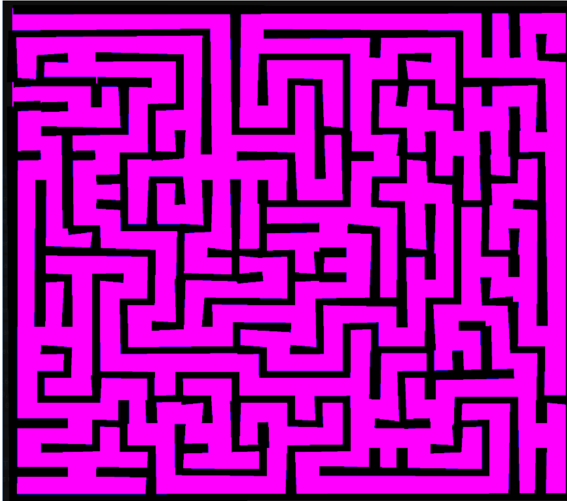

Figure A.3: This figure shows the GINT map of the labyrinth testing scenario at 2048 × 2048 GINT resolution.

Table A.1: HALOS pathfinding algorithm execution times. GINT resolution 128 × 128. Dynamic game world. Desktop NVIDIA.

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.116	2.41	81.202
City	0.113	0.514	78.405

Table A.2: HALOS pathfinding algorithm execution times. GINT resolution 512 × 512. Dynamic game world. Desktop NIVIDA.

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.159	3.805	90.315
City	0.158	0.889	98.892

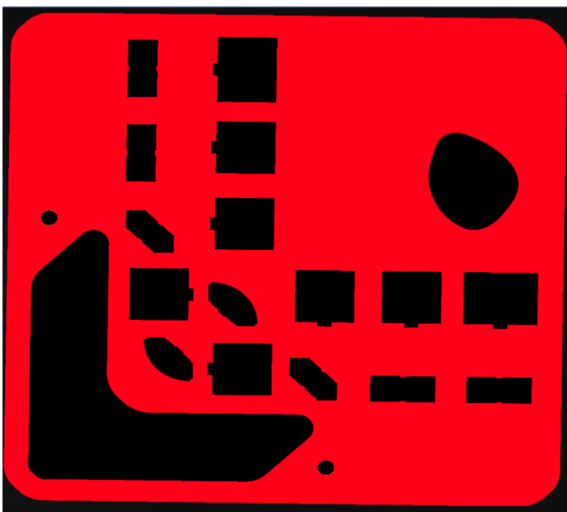

Figure A.4: This figure shows the GINT map of the vivianna testing scenario at 2048 × 2048 GINT resolution.

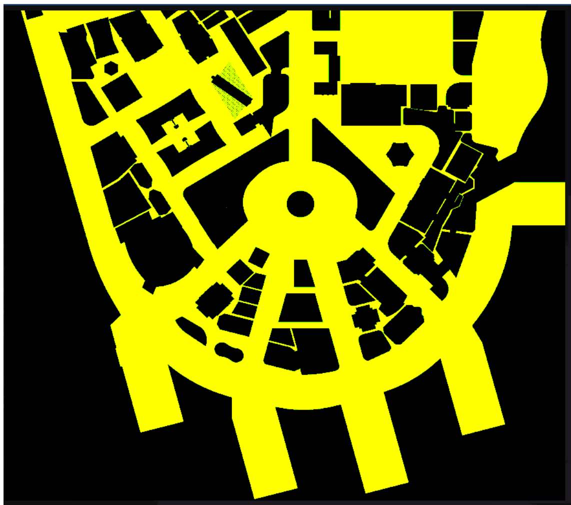

Figure A.5: This figure shows the GINT map of the castelia testing scenario at 2048×2048 GINT resolution.

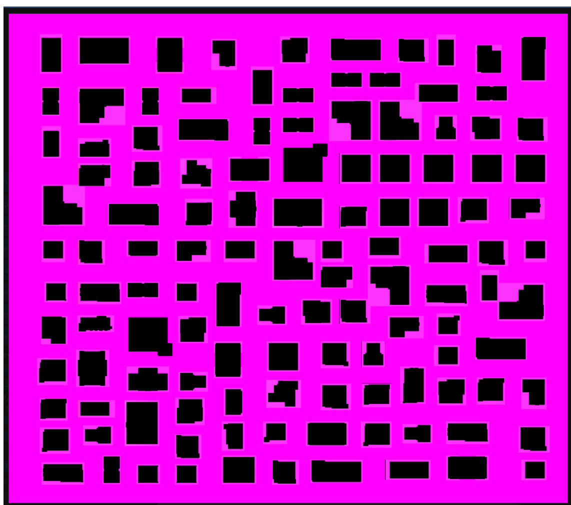

Figure A.6: This figure shows the GINT map of the city blocks testing scenario at 2048×2048 GINT resolution.

Table A.3: HALOS pathfinding algorithm execution times. GINT resolution 128×128. Dynamic game world. Desktop AMD.

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.367	7.36	13.183
City	0.285	1.463	3.389

Table A.4: HALOS pathfinding algorithm execution times. GINT resolution 512×512. Dynamic game world. Desktop AMD

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.302	11.38	20.368
City	0.326	2.946	6.039

Table A.5: HALOS pathfinding algorithm execution times. GINT resolution 128×128. Dynamic game world. Laptop AMD.

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.446	8.32	12.331
City	0.456	1.702	3.095

Table A.6: HALOS pathfinding algorithm execution times. GINT resolution 512×512. Dynamic game world. Laptop AMD

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.455	13.14	21.842
City	0.540	4.432	5.760

Table A.7: HALOS pathfinding algorithm execution times. GINT resolution 128×128. Dynamic game world. Laptop NVIDIA

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.087	1.631	39.988
City	0.085	0.317	8.383

Table A.8: HALOS pathfinding algorithm execution times. GINT resolution 512×512. Dynamic game world. Laptop NVIDIA

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.091	2.655	9.384
City	0.09	0.613	21.554

Table A.9: The execution times of the automatic GINT boundary calculation on two scenarios with varying GINT resolutions. NVIDIA desktop.

Testing scenario	GINT auto min (ms)	GINT auto average (ms)	GINT auto max (ms)
Labyrinth 128x128	0.162	0.211	9.531
Labyrinth 512x512	0.147	0.184	3.507
City 128x128	0.544	0.595	32.373
City 512x512	0.422	0.455	31.660

Table A.10: The execution times of the GINT navigation mesh generation on two scenarios with varying GINT resolutions. NVIDIA desktop.

Testing scenario	GINT min (ms)	GINT average (ms)	GINT max (ms)
Labyrinth 128x128	0.003	0.057	1.329
Labyrinth 512x512	0.019	0.076	0.733
City 128x128	0.003	0.064	3.082
City 512x512	0.028	0.069	6.273

Table A.11: The execution times of HALOS on two scenarios with varying GINT resolutions. NVIDIA desktop.

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth 128x128	2.392	3.457	26.700
Labyrinth 512x512	3.558	4.970	112.997
City 128x128	0.311	0.423	27.576
City 512x512	0.707	0.877	82.757

Table A.12: A table showing the attributes that signify precision. Not that if at least one of the agents does not reach their goal their collective traversal time becomes infinite. A world length unit is the same as a translation of 1.0 in world space.

Resolution	All agents reached the goal?	Collective traversal time (Seconds)	Collective Tile Distance (Pixels)	Collective Path Distance (Pixels)	Pixel density adjusted length (World)
32 x 32	75/80	Infinite	585	1295	16744
64 x 64	79/80	Infinite	1352	2006	13215
96 x 96	77/80	Infinite	1821	2502	11076
128 x 128	77/80	Infinite	2512	2847	9516
192 x 192	77/80	Infinite	3594	2929	8831
256 x 256	79/80	Infinite	4479	4905	8314
352 x 352	80/80	654	6429	7248	8977
448 x 448	80/80	664	7757	8456	8255
512 x 512	80/80	652	9588	9668	8268
640 x 640	80/80	685	14057	13045	8912
800 x 800	80/80	698	16287	16305	8923
1024 x 1024	80/80	696	22272	20839	8919
1440 x 1440	80/80	647	31317	28607	8719
1888 x 1888	80/80	699	33461	35297	8235
2048 x 2048	80/80	920	37215	38279	8235
2400 x 2400	80/80	2652	42517	42794	8046
2592 x 2592	80/80	2419	45922	47295	8047
2656 x 2656	27/80	Infinite	47053	48457	8047
2720 x 2720	1/80	Infinite	49544	49624	8046
3072 x 3072	1/80	Infinite	54431	56044	8048
3520 x 3520	0/80	Infinite	62344	64180	8045
4096 x 4096	0/80	Infinite	72561	74685	8046

Table A.13: HALOS pathfinding algorithm execution times. GINT resolution 2048×2048. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	17.133	21.247	108.208
Maze	4.374	5.331	70.392
Vivianna	3.426	4.133	62.547
Castelia	22.137	26.493	122.823
Quad	2.914	3.46	71.457
City	3.592	4.39	82.216

Table A.14: HALOS pathfinding algorithm execution times. GINT resolution 1024×1024. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	6.838	7.791	101.154
Maze	1.843	2.080	19.396
Vivianna	1.536	1.536	73.489
Castelia	5.099	6.341	30.099
Quad	0.924	1.063	11.459
City	1.425	1.601	12.067

Table A.15: HALOS pathfinding algorithm execution times. GINT resolution 512×512. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	3.158	4.22	251.237
Maze	0.778	1.043	81.82
Vivianna	0.604	0.782	79.858
Castelia	2.327	2.902	144.31
Quad	0.385	0.488	79.726
City	0.583	0.727	329.863

Table A.16: HALOS pathfinding algorithm execution times. GINT resolution 256×256. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	2.404	3.473	75.519
Maze	0.514	0.763	73.596
Vivianna	0.363	0.503	29.290
Castelia	0.931	1.368	31.630
Quad	0.236	0.312	21.796
City	0.403	0.566	40.720

Table A.17: HALOS pathfinding algorithm execution times. GINT resolution 128×128. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	2.392	3.457	26.700
Maze	0.442	0.613	13.067
Vivianna	0.322	0.463	21.412
Castelia	0.703	1.109	39.866
Quad	0.188	0.268	30.249
City	0.311	0.423	27.576

Table A.18: HALOS pathfinding algorithm execution times. GINT resolution 64×64. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	1.176	1.708	31.764
Maze	0.412	0.571	34.269
Vivianna	0.288	0.431	38.824
Castelia	0.776	1.151	29.434
Quad	0.179	0.236	9.695
City	0.311	0.444	19.290

Table A.19: HALOS pathfinding algorithm execution times. GINT resolution 32×32. NVIDIA Desktop

Testing scenario	HALOS min (ms)	HALOS average (ms)	HALOS max (ms)
Labyrinth	0.410	0.571	13.089
Maze	0.395	0.567	18.117
Vivianna	0.279	0.401	14.668
Castelia	0.641	0.975	34.772
Quad	0.180	0.280	18.319
City	0.351	0.533	26.633

Table A.20: Time measurements of a variable amount of agents on the Quad testing scenario.

Agents	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
0	0.207	0.296	31.340
1	0.203	0.361	48.822
64	0.206	0.356	47.897
128	0.277	0.347	35.030
256	0.220	0.412	43.584
512	0.226	0.752	78.053
1024	0.231	0.965	66.695
2048	0.277	0.906	38.736

Table A.21: Time measurements of generating the static GINT map on the scenarios Labyrinth and City on the GINT resolutions 128x128 and 512x512.

Scenario	Resolution	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
Labyrinth	128x128	0.003	0.057	1.329
Labyrinth	512x512	0.019	0.076	0.733
City	128x128	0.003	0.064	3.082
City	512x512	0.028	0.069	6.273

Table A.22: Time measurements of generating the dynamic GINT map on the scenarios Labyrinth and City on the GINT resolutions 128x128 and 512x512.

Scenario	Resolution	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
Labyrinth	128x128	0.003	0.052	2.142
Labyrinth	512x512	0.019	0.069	1.207
City	128x128	0.004	0.073	16.580
City	512x512	0.029	0.072	0.971

Table A.23: Time measurements of computing the dynamic automatic GINT boundary on the scenarios Labyrinth and City on the GINT resolutions 128x128 and 512x512.

Scenario	Resolution	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
Labyrinth	128x128	0.162	0.211	9.531
Labyrinth	512x512	0.422	0.184	3.507
City	128x128	0.544	0.595	32.373
City	512x512	0.422	0.455	31.660

Table A.24: Time measurements of computing the HALOS algorithm on the scenarios Labyrinth and City on the GINT resolution 512x512.

Scenario	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
Labyrinth	3.558	4.970	112.997
City	0.707	0.877	82.757

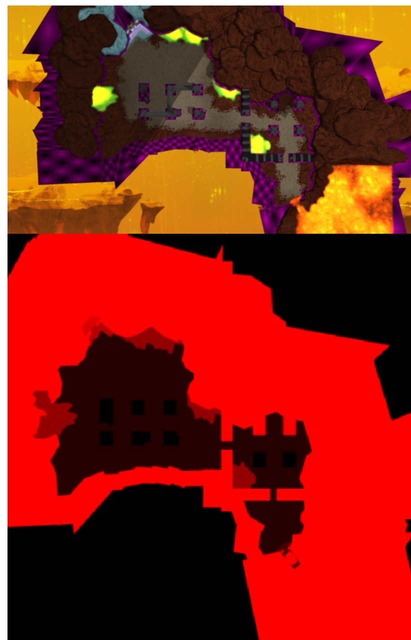

Figure A.7: An overview of a stage from a previous project making use of a prototypical version of GINT. The upper image is the 3D model of the stage whilst the lower image is the resulting GINT generated when the stage is put into the GINT rendering pipeline. The black and bright red indicate various types of walls whilst the darker shades of red represent different types of floors (solid ground, lava, acid, and ice).

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden