Real-Time Physics-Based Destruction with Scalar Field Terrains

Jordan Brown

Abertay University

School of Arts, Media and Computer Games

May 2016

ABERTAY UNIVERSITY

PERMISSION TO COPY

Author:	Jordan Brown
Title:	Real-Time Physics-Based Destruction with Scalar Field Terrains
Qualification:	BSc. (Hons) Computer Games Technology
Year:	2016

I certify that the above mentioned project is my original work.

I agree that this dissertation may be reproduced, stored or transmitted, in any form.

Signature: …….

Date: …….

Abstract

With an increasing demand for games which give players the ability to manipulate their surroundings freely, and the recent advances in graphics processing capability, finding efficient ways to model and simulate complex 3D terrains has become a topic of interest.

This project attempts to find the most efficient method to simulate physics-based destruction on these complex terrains.

By implementing several different methods of simulating terrains, combining them with several separate techniques, and analyzing their relative performance, the most efficient method can be determined.

Performance analysis graphs are displayed and explained in the results section, with discussion on the relative advantages and disadvantages of the most efficient terrain representation method.

Advances in graphics processing hardware power and architecture have led to previously inefficient methods becoming much more viable, such as *Distance-Based Raymarching.*

Foreword

I would like to thank Dr. Craig Stark for his mentorship, advice, and support throughout the development of this project.

I would also like to thank Iain Donald, Paul Robertson, Grant Clarke, Lynn Parker and Adam Sampson for their excellent mentorship, sound advice, and tutoring throughout my time at University.

I would like to thank my colleagues Matthew Cormack and Caspar Starton, for their close and continued support over many years.

Finally, I would like to give thanks to the Abertay Game Development Society and its members; their creative spirit is a shining example of the University's greatest strengths.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

Introduction 1

1. Literature Review 4

1.1 – Background / Context 4

1.2 – Existing Techniques & Evaluations 5

1.2.1 – Terrain Representation 5

1.2.2 – Physics Object Fracture 10

1.2.3 – Object to Terrain Reconstitution 13

1.3 – Evaluation of Current Techniques 14

2. Methodology 15

2.1 – Application Structure 15

2.2 – Terrain Representation 17

2.3 – Mesh Fracturing 22

2.4 – Terrain Recombination 25

2.5 – Performance Analysis 27

3. Results 29

3.1 Grid-Based Performance 30

3.2 – Distance-Based Raymarching Performance 34

3.3 – Event Logging & Overall Performance 39

4. Discussion & Conclusion 46

4.1 – Critical Evaluation 46

4.2 – Future Work 47

4.3 – Conclusion 48
References 49
Bibliography 53
APPENDIX A: Application Images 55

LIST OF TABLES

Table 1: System Specifications 29

LIST OF FIGURES

Figure 1. Minecraft. (Mojang, 2015) 1

Figure 2. Space Engineers. (Keen Software House, 2016) 1

Figure 3. Floating Terrain; "massive floating mountains", (Minecraft Seeds, 2011) 2

Figure 4. Accuracy Issues with Marching Cubes, (Ronen Tzur, 2004) 6

Figure 5. Marching Cubes Hard Edges Diagram, (Ronen Tzur, 2004) 6

Figure 6. Dual Contouring (Ronen Tzur, 2004) 7

Figure 7. Dual Contouring Diagram (Ronen Tzur, 2004) 7

Figure 8. Distance-Based Raymarching Scene, "slisesix" (Iñigo Quilez, 2008) 9

Figure 9. Voronoi-Fractured Object (Sara C. Schvartzman et al, 2014) 11

Figure 10. Red Faction (Red Faction Wikia, 2009) 14

Figure 11. Application Structure 16

Figure 12. 2D Slice of Grid (B. Anderson, 2016) 18

Figure 13. Distance-Based Raymarching Diagram 20

Figure 14. Target Voronoi Cell Shape (Esteve, J. 2011) 23

Figure 15. Recursive Mesh Slicing (Esteve, J. 2011) 23

Figure 16: Average Rendering Frame-Times for Grid-Based Naive Marching Cubes Parameters 31

Figure 17: Average Physics-Update Frame-Times for Grid-Based Naive Marching Cubes Parameters 32

Figure 18: Average Memory Usage for Grid-Based Naive Marching Cubes Parameters 33

Figure 19: Application in Grid-Based Mode, at full grid resolution 35

Figure 20: Application in Distance-Based Raymarching Mode, at full resolution 35

Figure 21: Average Rendering Frame-Times for Distance-Based Raymarching Parameters 36

Figure 22: Average Physics-Update Frame-Times for Distance-Based Raymarching Parameters 37

Figure 23: Average Memory Usage for Distance-Based Raymarching Parameters 38
Figure 24: Gradual Performance Degradation after Repeated Carve Operations in Raymarch mode 41
Figure 25: Performance returns to baseline after switching to Grid-Based mode 42
Figure 26: Performance Spikes in Render Frame-Times when Recalculating Physics Mesh 43
Figure 27: Absence of spikes in Physics Update Frame-Times when Recalculating Physics Mesh 44
Figure 28: Performance impact reduced to baseline after objects re-combine with terrain 45
Figure 29: Project Application, Grid Mode, Low Grid Resolution 55
Figure 30: Project Application, Grid Mode, Low Grid Resolution with Physics Overlay 55
Figure 31: Project Application, Grid Mode, High Grid Resolution 56
Figure 32: Project Application, Raymarching Mode, Full Resolution 56
Figure 33: Project Application, Raymarching Mode, Full Resolution with Physics Overlay 57
Figure 34: Project Application, Raymarching Mode, Full Resolution with Physics Overlay and Render Distance Increased to Maximum 57
Figure 35: Project Application, Grid Mode, Tunnel in Terrain 58
Figure 36: Project Application, Raymarching Mode, Tunnel in Terrain 58

Introduction

There is currently, in both the games industry and the consumer community, a growing interest in sandbox games. Sandbox games are games in which the player is given no specific instructions, but instead a wide range of tools and a modifiable world, in which they can do as they like within the confines of the game's mechanics. Some chief examples of currently-available sandbox games include "*Minecraft*", and "*Space Engineers*". *(Minecraft, 2009)(Space Engineers, 2013)*

Figure 1. Minecraft. (Mojang, 2015)

Figure 2. Space Engineers. (Keen Software House, 2016)

Many of these sandbox games contain tools for destruction as well as for construction; this is made possible by the games' usage of density field terrains. Density fields, also known as scalar or voxel fields, are used to represent volumetric or three-dimensional data, not dissimilar to how pixels are used to represent two-dimensional data.

These fields can represent any three-dimensional space or object, provided they are large enough and the information can be represented mathematically. Terrains are an ideal use-case, as they generally can be represented in these fields as a surface or set of surfaces. The volumetric nature of these fields means that certain terrain features (like caves or overhanging cliffs) can be represented fully, unlike simpler forms of terrain rendering which rely on height information only. This provides a high degree of freedom for players. However, current implementations of these scalar-field terrains do not simulate the terrain in a physical way. Terrain geometry is added to and removed from without equivalent exchange of mass; the destruction mechanics in these games often involve large chunks of the terrain simply disappearing, rather than being displaced or broken up.

Figure 3. Floating Terrain; "massive floating mountains", (Minecraft Seeds, 2011)

The reason that these terrains are not physically simulated is that the performance cost of simulating such a large amount of terrain would simply be too high, and as such, it is ignored. Physics objects in games are usually simple rigid-body objects, numbering relatively few compared to the areas of terrain in which they move. However, it would be possible to take areas of the scalar-field terrain, transform the areas into many physics

objects, simulate them, and then reconstitute them back into the terrain. This would allow for physically-simulated destruction and displacement of the terrain without significantly increasing processing cost.

The question this project aims to answer is then, effectively, “What is the most efficient method of simulating real-time, physics-based destruction when dealing with scalar field terrains?” To answer this question, one must investigate and construct a system in which realtime physical destruction of a scalar-field terrain can take place, while analysing its effectiveness and performance.

In doing so, the following objectives must be completed;

• Research and understand the algorithms/methods necessary to represent a scalar-field terrain, those necessary to transform scalar-field terrain chunks into one or multiple physics objects, and those necessary to reconstitute the objects back into the terrain.

• Construct the system, and an application in which to manipulate it, using multiple variants of the algorithms and techniques learned as above, in the primary areas of representing the terrain, fracturing the terrain, and reconstitution of the terrain.

• Analyse the performance of various combinations of techniques. Deduce which combinations are appropriate for various operating parameters (e.g, which would be best for a mobile games platform vs a desktop computer).

If these objectives are successfully completed, the resultant system could theoretically be built into any currently existing games engine.

1. Literature Review

1.1 - Background/Context

Scalar-field terrains have been explored in many contexts both inside and outside of the games industry. They have a prevailing presence in papers presented at the annual SIGGRAPH conferences. SIGGRAPH are *"an international community of researchers, artists, developers, filmmakers, scientists and business professionals who share an interest in computer graphics and interactive techniques." (SIGGRAPH, 2015)*

These papers cover a range of topics, and those focusing on scalar fields tend towards explaining common methods for constructing, organising, traversing, and representing these fields in software in a general sense, without specifically applying them to games. However, the research from these papers is invaluable when attempting to use scalar fields in a games-development context, as the algorithms and methodologies that have been developed often focus on performance as well as accuracy, and performance is paramount in successful games software.

It is for this reason that a large majority of the papers discussed in this section will be sourced from SIGGRAPH conferences.

The project can be divided into three separate components; terrain representation, physics object fracture, and object to terrain reconstitution. Papers relevant to each of these three topics will be discussed below.

1.2 - Existing Techniques & Evaluations

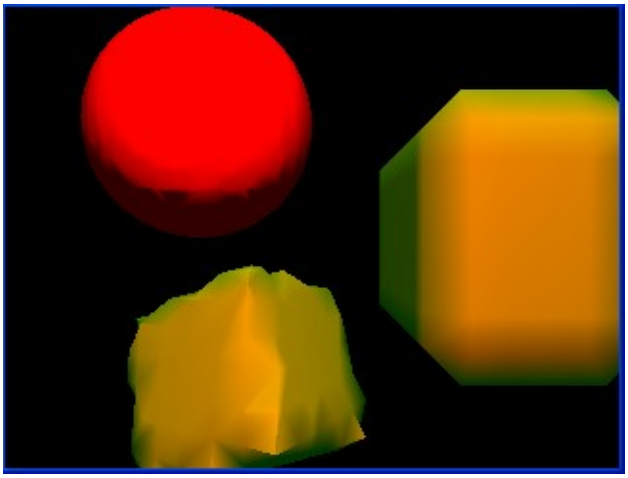
1.2.1 - Terrain Representation

One of the most popular and enduring techniques for representing scalar field data is known as the “Marching Cubes Algorithm”. This technique was first presented in the paper “*Marching Cubes: A High Resolution 3D Surface Construction Algorithm*” (*Lorensen & Cline, 1987)* at SIGGRAPH 1987. The technique was conceived as part of a research effort to visualize information from medical scans, and uses a table of pre-calculated polygonal shapes to represent the terrain. Selecting which shape to use is done by determining which vertices of a cube are inside or outside of the surface, where the cube is part of a grid of cells which cover the visible region of the terrain. The polygonal shapes are then interpolated along the curvature of the surface, so that the terrain appears smooth. The exact algorithm is patented, and so most implementations today are based on the similar but slightly more efficient approach by Paul Bourke, outlined in his 1994 paper “*Polygonizing a Scalar Field” (Bourke, 1994)*. While not exactly the same, the name “Marching Cubes” is often attributed to Bourke's implementation. This paper will henceforth refer to Bourke's implementation as such.

Marching Cubes is a highly efficient approach to this problem, provided the terrain does not change often. The algorithm is somewhat parallelisable; that is, the operations involved in solving the algorithm consist of many non-consecutive repetitions, and those repetitions can be done simultaneously instead of in sequence. By doing so, the algorithm becomes performant enough to be used in a scenario where the terrain can be in a state of motion. Parallel processing is something that modern graphics processors (GPUs) are extremely proficient at, and so Marching Cubes is a popular algorithm to use in general purpose GPU programming. The “geometry shader” stage in the programmable graphics pipeline is a primary example of where this algorithm becomes most performant; this stage of graphics processing takes vertices as inputs, and can create additional geometry as output. Given

that the algorithm consists of checking the status of a vertex and then creating additional geometry depending on some factor, it is the ideal environment in which to run the algorithm.

Marching Cubes is not without its downsides, however. As shown in *Figure 4*, the algorithm suffers from accuracy problems; hard edges along the sides of cube shapes suffer from visible distortions as they cross voxel boundaries.



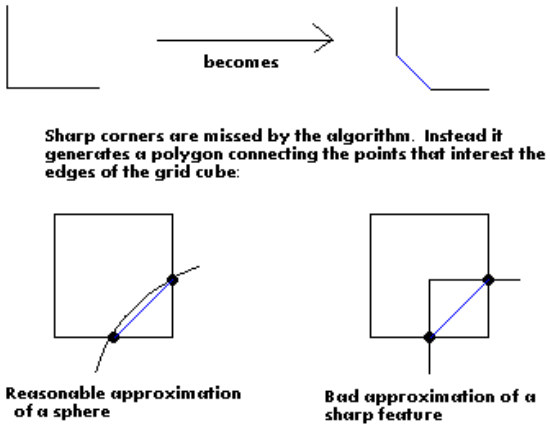


Figure 4. Accuracy Issues with Marching Cubes, (Ronen Tzur, 2004)

Figure 5. Marching Cubes Hard Edges Diagram, (Ronen Tzur, 2004)

There exist slightly less performant, but much more accurate algorithms that can represent such terrain features correctly. One such algorithm is known as the “Dual Contouring” method, which was first presented at SIGGRAPH 2002 in a paper titled “*Dual Contouring of Hermite Data” (Tao Ju, et al, 2002)*. The paper describes a method based on an extended form of Marching Cubes where instead of a uniform grid of cells being processed, the scalar field is organized into an octree structure, and each vertex contains additional information that lists the vertex's normal to the surface contour.

By examining the normals during the geometry processing stage, it can be determined if the points are part of a smooth surface or part of a hard-edged surface, allowing a much more accurate approximation of the shape. This can be seen in *Figure 6*, where unlike the image shown in *Figure 4*, the cuboid shapes are represented in sharp detail.

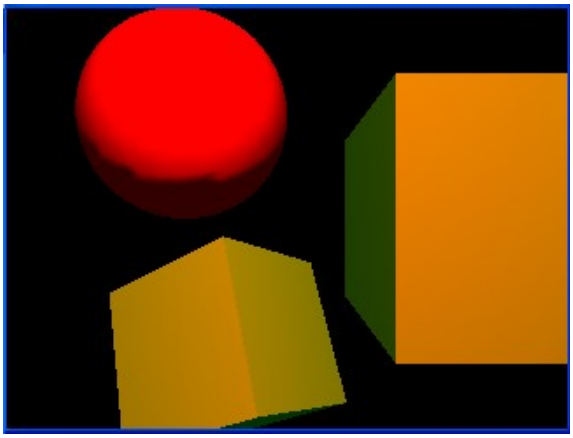


Figure 6. Dual Contouring (Ronen Tzur, 2004)

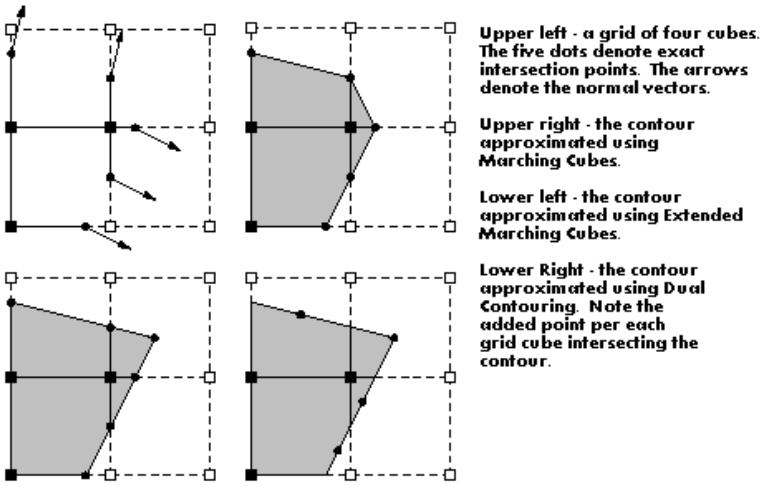


Figure 7. Dual Contouring Diagram (Ronen Tzur, 2004)

If the scalar field data were organized in a grid like the Marching Cubes algorithm, Dual Contouring would likely be outperformed. However, due to the octree structure, a great deal of processing can be skipped; octrees are recursively nested structures, allowing for more or less resolution based on various selective properties (in the case of computer

graphics, this is usually distance to the viewer – closer objects have higher resolution, whereas those far away usually have much lower resolution, to save unnecessary processing) in order to reduce computational complexity. While there is some processing overhead in managing the structure of an octree, this is far outweighed by the processing time saved by such operations. Effectively, the Dual Contouring method represents the terrain at a much greater detail in exchange for a more complex data arrangement. This can make it more difficult to generate or build a terrain, but allows for a much more engaging experience.

Both of these methods rely on polygonization for their end result; however, there are other methods which can represent complex terrains. These involve the use of a path-tracer or ray-tracer. Ray-tracers are methods in which the scene is rendered by physically simulating rays of light backwards from the viewer's position, towards the scene, bouncing off objects and finding light sources. These renders are usually highly accurate, but can take many minutes (even hours, days, or weeks) for a single frame. As such, they are not appropriate for games software in most cases – but there are exceptions.

By nature, the problem is very easily parallelizable – modern graphics hardware has reached a stage where some ray-tracing algorithms run in real-time, doubly so if they use optimization techniques.

One such technique is known as distance-field raymarching. In this system, the rays of light are not simulated fully, instead stepping forward in discrete amounts and only checking for collisions at those points. Ordinarily, stepping in discrete amounts may lead to the rays of light passing through objects thinner than the step distance, and would thus be extremely inaccurate. However, the distance-field system prevents this.

Distance fields are simply scalar fields where the values in the field represent not just whether or not terrain is present, but how far the closest point on the surface is from the current point in the field.

By stepping at or just below the distance to the nearest surface each time with the ray, the algorithm can perfectly encompass all possibilities whilst also taking advantage of the significant performance advantages of not contiguously simulating the ray.

Figure 8. Distance-Based Raymarching Scene, "slisesix" (Iñigo Quilez, 2008)

This technique was brought into the graphics programming scene with a presentation titled "*Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes" (Iñigo Quilez, 2008)* at a graphics conference called *"nvscene".*

The performance of the technique is relatively high for both the time period in which it was presented and with regards to other ray-tracing techniques. Additionally, the memory footprint of this rendering technique is extremely small, with huge terrains being represented with extremely small amounts of computer memory, many orders of magnitude below polygonization-type methods. However, the overall performance often fails to exceed that of polygonization methods, and as such has been largely put aside in favour of those methods.

1.2.2 - Physics Object Fracture

When simulating chunks of moving/loose terrain, the above methods are neither suited for nor easily modifiable for arbitrary translation and rotation of terrain features, and cannot be used to fully represent the physical state of a moving rigid-body object, as they are approximations of points in space. While they can represent moving objects in the surface, the object's shape and mass will continually shift as the algorithm attempts to resolve a technically infinitely-detailed or fractal surface. Instead, it is more performant to convert a selected area of terrain into a simple polygonal rigid body and simulate it with a standard solution.

However, for the destruction of terrain to appear convincing or effective, the terrain chunk must be fractured into smaller pieces; simply carving out and simulating the large, contiguous piece of displaced terrain would be highly unconvincing. Additionally, common problems like building destruction would not work in such a case, as the building would remain whole and tumble around in one piece.

This fracturing of terrain has been attempted successfully before in games, even on mobile platforms (*Gustaffson, D. 2014*) without a noticeable drop in performance. It does, however, take many forms – this paper will focus on one particular form.

Usually, when games attempt mesh fracturing, they use pre-computed fracture meshes for the objects; these always fracture in exactly the same way. They are extremely light on processing time as they have all been built in modelling software beforehand. However, they cannot adapt to changing circumstances or shape, and are therefore unsuitable for this project and its goals.

A good approximation of mesh fracturing in real-time can be accomplished through the use of Voronoi Diagrams. Voronoi Diagrams are partitions of a plane or space into regions based on distance to cell centers in the plane or space. They result in angular, multi-faceted divisions when applied to 3D volumes. This gives the general appearance of fractured glass or rock.

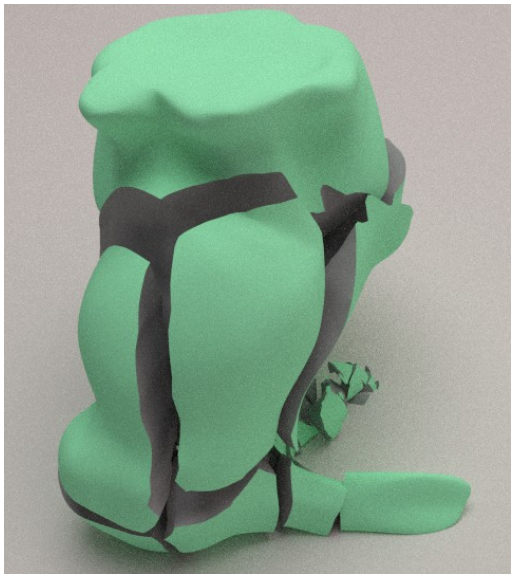


Figure 9. Voronoi-Fractured Object (Sara C. Schvartzman et al, 2014)

While not all terrain to be represented can be said to consist of solid rock, it is applicable to some scenarios, and tends to be highly performant at small cell counts.

Generally, the algorithm works as such: cell centres are scattered throughout the volume that should be fractured. The mesh of the object is then split using planes co-planar with the faces of each cell. After all the splits are done, the resultant meshes are given new rigid-bodies and allowed to continue physical simulation. For further accuracy, these points can be generated if the original body is impacted by another, and scattered based on inverse-square distance from the impact point. This gives the "spiderweb" effect seen on panes of glass, and is topologically consistent (if not exactly simulative) with the way rocks fracture in the real world.

If so desired, these points can be configured in a way that gives artists control over the way an object fractures; this and other techniques for preserving artistic intention (as well as an overview of the Voronoi method) can be found in the paper "*Physics-Aware Voronoi Fracture with Example-Based Acceleration" (Sara C. Schvartzman et al, 2014)*.

The computational impact of repetitive mesh slicing based on planes is, however, not

minute. As the number of cells increase, so do the number of slices that must be made – and with those, the number of vertices in each resultant mesh. A high-resolution voronoi diagram can result in meshes with vertex counts many orders of magnitude above the original, which has an impact not only on further processing of the mesh, but also rendering and physics calculations. It is therefore wise to simplify the meshes after such recursive operations, either by approximating the surface by using its outermost points (the “Convex Hull” methodology) or by selectively reducing similar vertices (the “Merge” methodology). Both trade accuracy of the mesh for performance, and can be unreliable when presented with concave shapes.

Another, very similar method, can do away with these steps in exchange for slightly more complex starting conditions. By shaping the boundaries of the Voronoi Diagram to conform with the original mesh, the divisions of the cells in the voronoi diagram can be used as the resultant meshes themselves, without having to recursively perform mesh-slicing operations. However, creating shaped Voronoi Diagrams requires forming computational boundaries where the diagram cannot be generated, and so introduces a new problem to be solved.

1.2.3 - Object to Terrain Reconstitution

This stage has considerably less literature surrounding it – converting a polygonal object (or polyhedron) back into scalar-field information is not needed as often, as artists would simply just build the object as a polygonal object. There are, however, still papers which discuss this issue. The 2011 paper "*BSP-fields: An Exact Representation of Polygonal Objects by Differentiable Scalar Fields Based on Binary Space Partitioning" (O. Fryazinov, et al, 2011)* discusses the specifics of converting generic polygonal objects back into that form with a focus on producing 1:1 exact representations of objects whilst preserving computational continuity of the function. While almost perfectly accurate, his method is highly complex, and very computationally expensive. The performance impact is mentioned as being particularly high in the paper itself, and as such it is not suitable for the purposes of game-type applications.

Instead, a much more performant (but less accurate) method is outlined in "*Single-Pass GPU Solid Voxelization for Real-Time Applications" (E. Eisemann & X. Décoret, 2008).* It works similarly to medical scanners; by taking a series of sliced renders of a polygonal object, the GPU can create a rough scalar field map of the object. This is easily done in real-time, as the slices can be computed in parallel on the GPU. This is prone to losing accuracy, but suffices for real-time game applications, with a median time of 0.26ms for objects with roughly 5,000 triangles (15,000 vertices). Given that the voronoi-sliced meshes will be of smaller size, and that the algorithm will run on occasion as the objects come to rest, it outperforms any similar solution by a huge margin.

1.3 - Evaluation of Current Techniques

The techniques discussed above have been applied with varying degrees of success in both the recent and further past; one of the most earliest examples of note would be the "*Geo-Mod"* technology used in Volition's "*Red Faction" (Red Faction, 2001).* The game allowed players to tunnel through volumes of rock with explosives and drilling machines. While the focus of the game was not on the terrain manipulation, it played an important part in making the combat gameplay distinctive, allowing players to approach their opponents from unconventional locations.

Figure 10. Red Faction (Red Faction Wikia, 2009)

These modifiable areas of terrain were in small, enclosed volumes, but were responsible for much of the game's popularity. A more modern example is from the previously-mentioned "*Space Engineers" (Space Engineers, 2013)* in which players can build spacecraft and drill into floating asteroids and planets modelled with scalar-field terrains – however, the residual pieces of terrain mined from the planetoids can remain fixed in place as though they were still part of the original body.
As such, for a technology with a relatively long history, it remains largely unchanged.

2. Methodology

There are five major segments of this project's implementation; *Application Structure, Terrain Representation, Mesh Fracturing, Terrain Recombination* and *Performance Analysis.* These segments will be discussed in detail, in order.

2.1 - Application Structure

The design of the application's structure is of relatively high importance; in order for the techniques discussed above to work as a cohesive whole, the framework in which they co-exist must be carefully planned.

This application is written in C++, and makes use of the OpenFrameworks software library. OpenFrameworks is *"an open source C++ toolkit for creative coding" (OpenFrameworks, 2016).* It focuses on having a modular architecture, acting as a sort of bridge between different libraries. Its primary use in this case is as an abstraction layer for the *OpenGL* graphics API, and as an interface for the *Bullet Physics Library,* a feature-complete and tested implementation of Newtonian physics.

These libraries and frameworks provide a stable base upon which to build the rest of the application.

The application is essentially a small sandbox allowing a user to view a 3D scene containing the terrain and a small number of tools for manipulating the terrain and viewing live performance statistics.

The user may move the camera around the scene as they wish, and their primary action upon the terrain will be causing sphere-shaped segments to be cut away. These sphere-shaped segments will be converted into physics objects, fractured into several more objects, and have forces applied to represent an explosion on the terrain's surface. Once the pieces come to rest, they will be re-absorbed into the terrain.

The user may select multiple different methods and tweak various settings for displaying the terrain and fracturing the meshes, in order to see the performance and quality differences.

The application is designed such that the different methods can be toggled on and off during operation without requiring the user to restart. This is accomplished by building the software in a modular fashion, in which each major component of the system is encapsulated entirely within interchangeable blocks of code.

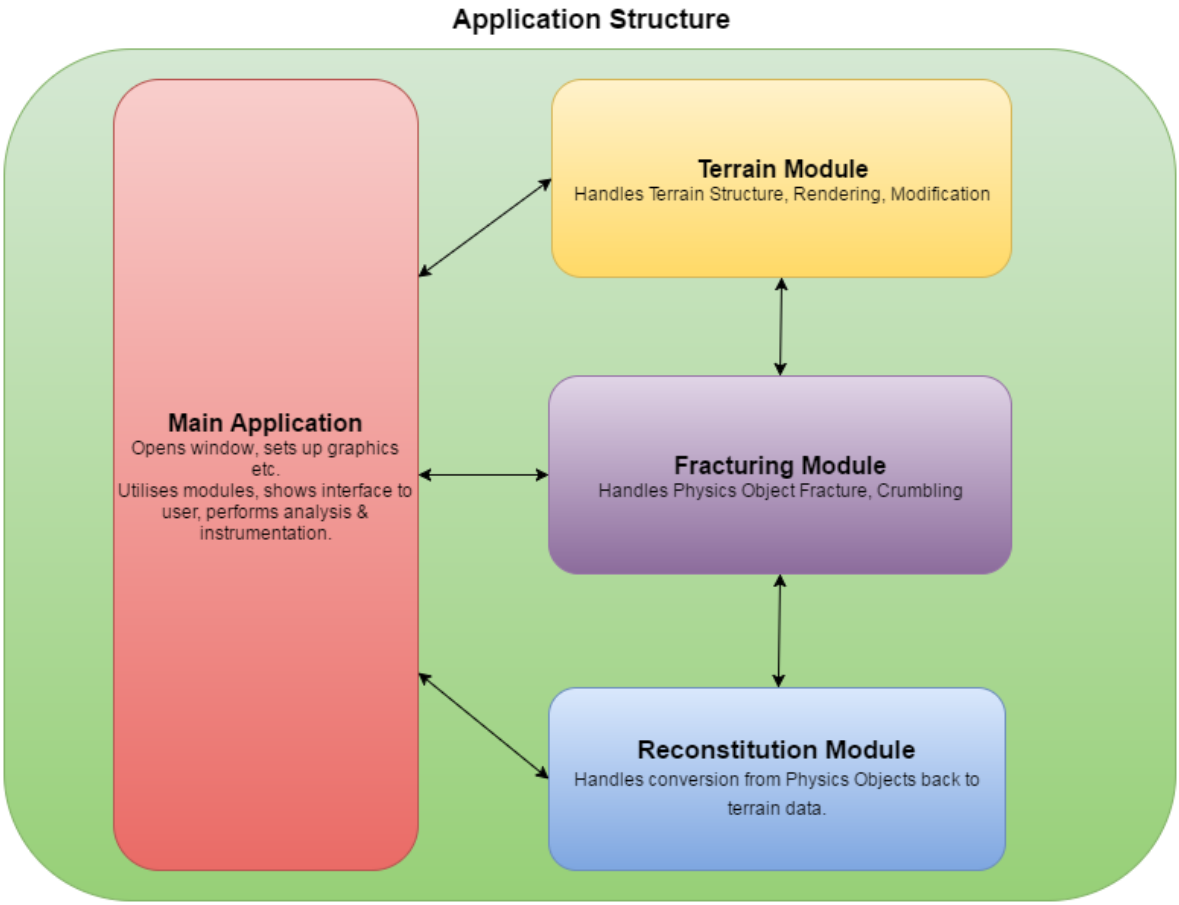


Figure 11. Application Structure

2.2 - Terrain Representation

This segment is concerned with how the implementation builds, organizes, stores, and renders a scalar-field terrain; in order for any manipulation of a terrain to be done, it must first exist.

There are multiple ways to display scalar-field terrains, as discussed previously; there are also multiple ways in which to represent the terrain in memory. In the interests of time and simplicity, this implementation chooses to represent the terrain in memory in just one way, while allowing for different rendering methods. The rendering methods are most likely to impact performance and are immediately obvious to the user when changed.

Due to time constraints, only two methods for rendering have been considered:

1. *Grid-Based Naive Marching Cubes*
2. *Distance-Based Raymarching*

Grid-Based Naive Marching Cubes is an implementation of the previously-discussed Marching Cubes algorithm that has been developed as part of this project. [images, diagrams?]

In *Grid-Based Naive Marching Cubes* (referred to from this point forward simply as *Grid-Based*), the application renders a uniform 3d grid of variable resolution and scale, as a set of points in 3D space. This grid moves as the user moves the scene's camera around, so that it gives the illusion of remaining fixed upon the screen. It does not rotate with the camera, so the user can observe all points of the grid.

This becomes useful for implementing Marching Cubes on the GPU; there is a stage in the programmable graphics pipeline called the *geometry shader,* which allows the GPU to programmatically generate additional polygons from a smaller input set. It performs these geometry shader operations on each input vertex simultaneously, in parallel. This means that for each point along the fixed grid, the Marching Cubes algorithm can evaluate the new polygons for a voxel in that position, and display them on screen. While this means that effectively the terrain is being re-calculated every frame, the advantages of the

parallel-processing on the GPU mean that there is relatively little performance impact compared to a software-based implementation. This allows for much larger or more detailed terrains that can be modified in real-time.

The terrain's shape is calculated from a density function evaluated in the geometry shader. A simple density function for a terrain can be obtained from sampling a *noise texture,* which can be made to repeat infinitely (giving an infinitely-sized terrain). The density function is also responsible for dealing with *Constructive Solid Geometry (CSG)* operations; *CSG* operations are simply boolean operations between density functions that represent primitives (for example, a sphere's density function is based on its position and radius) and the current terrain data. This allows for cumulative alterations to the terrain via addition and subtraction of field information.

Because the grid moves with the camera, the user can effectively explore the entire terrain without the complete terrain having to be rendered at once. Due to this, the scale and resolution of the grid can be tweaked to represent the same terrain on less performant hardware.

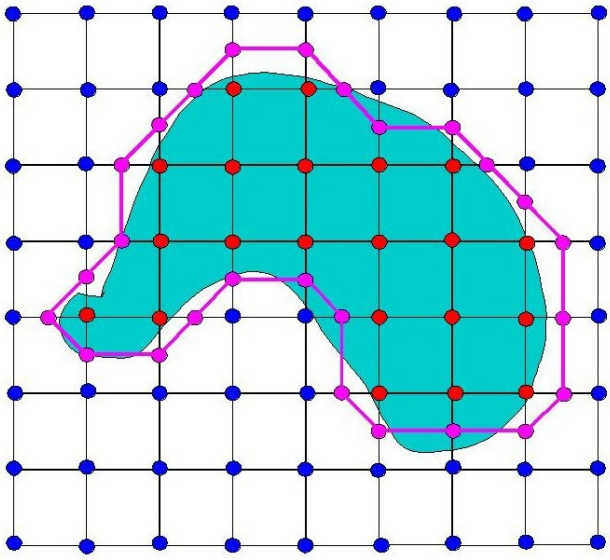


Figure 12. 2D Slice of Grid (B. Anderson, 2016)

The implementation is considered *naive* because it is relatively unoptimized; there are a great deal of changes that could be made to the implementation that will be discussed in detail in the evaluation section. One such optimization would involve the grid's spacing being non-uniform, having the points closer together in the center and getting progressively further apart as the edges of the grid were reached. This would mean that the terrain would have a higher precision and detail close to the user, while maintaining the same performance footprint. Distant objects would be imprecisely represented and odd, but would likely matter less.

The second rendering method, *Distance-Based Raymarching*, is a high-detail, high-precision method with a large performance footprint. Instead of using the *geometry shader*, all of the work for this method is done in parallel on the *fragment* or *pixel shader* stage, which handles choosing which colours to draw on-screen. The actual object rendered by the software is just a simple plane that covers the screen.

Raymarching works similarly to ray-tracing. Ray-tracing works by intersecting lines, or rays, from the camera's focal point with the mathematic surface of the terrain, which in this case, is the density field. Where the ray intersects the terrain, a pixel is drawn onto the screen. The pixel's colour depends on the angle that the ray strikes the terrain in relation to the position of the light source, mimicing (in reverse) how light moves in the physical world. With this techinque, the surface of the terrain can be represented absolutely correctly, and complex light calculations can be added to give the scene a much more realistic look. Ordinarily, this kind of simulation takes a long time to be processed, because of the complex integration mathematics involved. However, Raymarching checks for an intersection at set intervals along the ray, instead of attempting to solve numerically. This can cause precision issues, where the surface may be between the intervals, and the ray would simply step through, creating a hole.

The chief difference with *Distance-Based Raymarching* is that instead of stepping at a fixed rate, the simulation steps forward based on the value at that point of the terrain's density field; by knowing how far away the nearest area of terrain is, the simulation can both skip

large areas where there is no terrain and reduce its error as it approaches the terrain's surface. This combines the accuracy of the ray-tracing method with the performance gains from the Raymarching method.

As can be seen in *Figure 13,* the number of steps taken rises as the ray passes close to object boundaries (marked *a.* on the diagram) and reduces as the ray passes open spaces (*b.*). For shapes with many close boundaries this can present a performance problem.

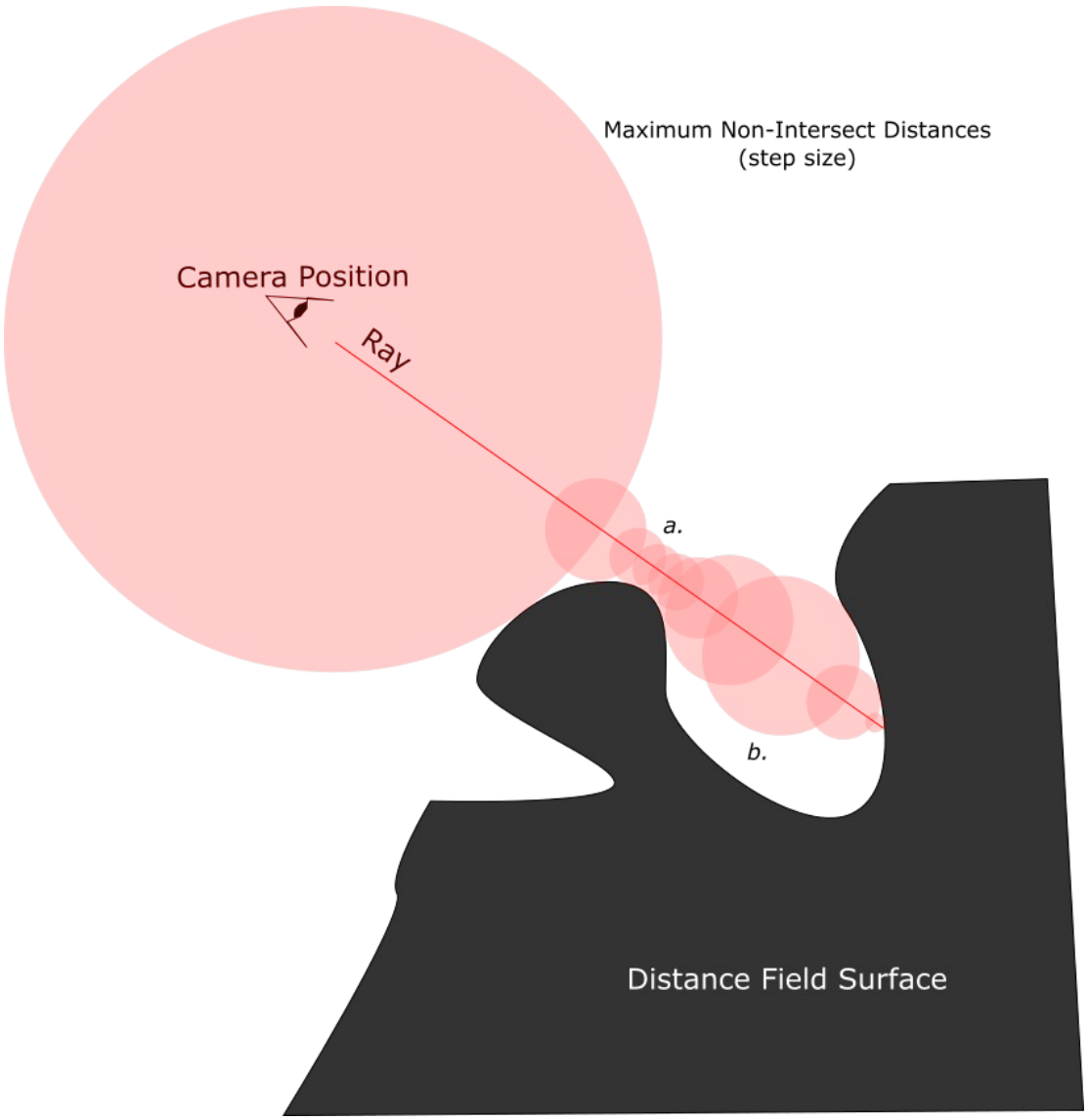


Figure 13. Distance-Based Raymarching Diagram

A single ray is fired for each pixel of the screen, and so the overall performance is entirely dependent on the resolution of the texture being rendered, which can be adjusted by the user.

It should be noted that the density function does not change significantly between this method and *Grid-Based,* as both use the same scalar-field information to generate their results. Because of this, even the *CSG* operations work similarly, saving a great deal of time when implementing these two methods.

However, the polygonization method in *Grid-Based* has a clear implementation advantage; by fetching the polygon information from the GPU, a physics mesh can be generated on the CPU; this allows the *Bullet Physics Engine* to use the terrain as-represented on screen to perform physics calculations and collisions for the moving physics objects that appear later. Without this, the raymarching method must use a primitive form of triangulation in which there are a few, sparse rays cast from above the terrain onto it, and a loose physics mesh is constructed from this information. As such, the physics information can be wildly inaccurate. Alternatively, each object could generate its own world physics mesh at each frame by firing a ray from its own position onto the terrain. This would cause a performance drop as the number of physics objects increased.

2.3 - Mesh Fracturing

This segment is concerned with how physics objects are extracted from the terrain, and how they can be broken into several more objects.

Firstly, *CSG* operations on the terrain are by nature boolean operations, such as AND, OR, and so on. By applying a boolean NOT to a combination of these operations, it is possible to receive not the altered terrain, but an inverted case where the terrain is gone, but the segment of terrain to be removed by the operation is left isolated. By extracting the geometry of this piece of terrain, a physics object can be generated of the same size, shape and dimensions. The application does this at the moment an alteration to the terrain is made. The resultant rigid body object is then subject to physical forces in the environment.

Next, this large object needs to be fractured into smaller pieces to give the illusion of realistic explosive destruction. These chunks will be done using the aforementioned *Voronoi Fracturing* method, wherein a *Voronoi diagram* is constructed around the shape and then recursive slices are made based on the planes that constitute each Voronoi cell within the diagram.

Because the application is using *openFrameworks,* it can make use of a module which provides a bridge to the *voro++* library *(voro++, 2016)* which allows the application to quickly compute and manipulate Voronoi diagrams.

Slicing a mesh is relatively simple; by intersecting the plane to cut by with every face of the mesh, it can be determined which vertices lie inside and outside of the plane. At the points of intersection, new vertices are generated and the resultant hole left behind is *capped,* in this case with an extremely simple method in which a vertex is generated in the centermost point of the hole, and the vertices on the edges of the hole are joined to it.

By performing this operation many times on a single shape, a fragment of that shape can be generated. This is done as many times as necessary to build fragments for each Voronoi cell. As such, it is not particularly performant, but is both reliable and accurate. Physics objects are then generated using the meshes of the fragment pieces, and the original shape

is replaced with the physics objects in their correct relative positions. The physics objects inherit the velocity of their parent shape, and can have forces applied to them to emulate explosive force. This gives the illusion of a large shape cracking and shattering into many smaller pieces.

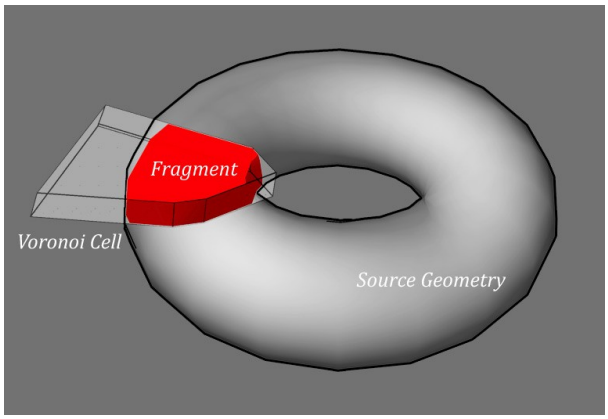


Figure 14. Target Voronoi Cell Shape (Esteve, J. 2011)

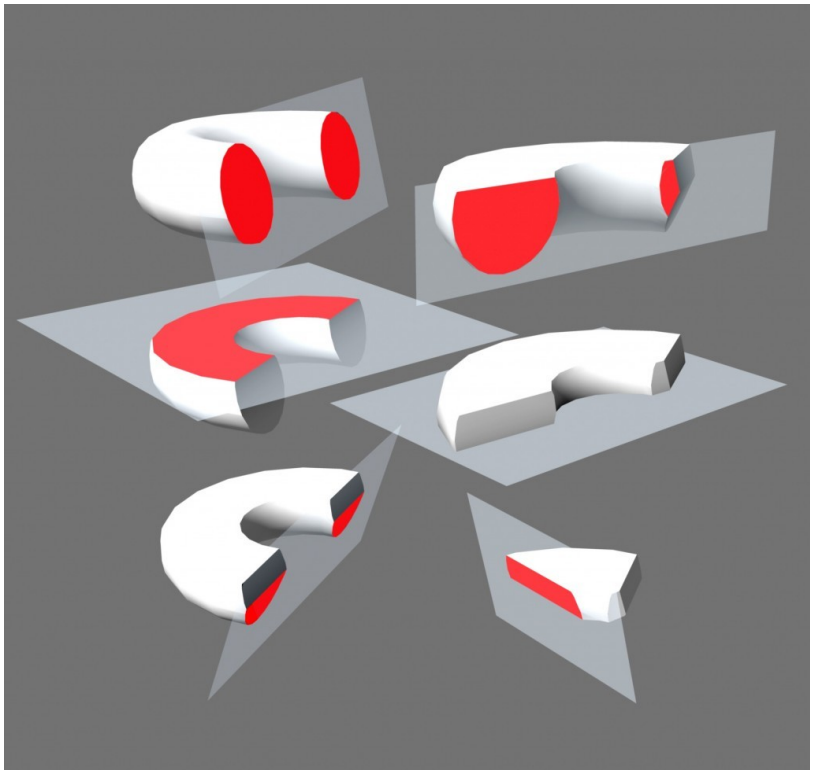


Figure 15. Recursive Mesh Slicing (Esteve, J. 2011)

Recursive slicing, however, has performance implications; by adding extra vertices (and thus, extra intersection points) every time a slice is made, the total number of resultant vertices per fragment rises exponentially. This can lead to memory allocation issues, processing time delays, and rendering problems. As such, care must be taken to simplify the meshes where possible, such as by removing vertices that are redundant.

2.4 - Terrain Recombination

This segment is the penultimate segment of the application, and is necessary for the performance aspect of this project; by merging the fractured pieces back into the density field, and deleting their physics objects, large amounts of terrain can be fractured and shifted around without continually increasing the physics simulation processing load.
This implementation will only use one of the methods previously discussed; the *Progressive Slice Voxelization Method* wherein the physics objects are rendered in slices to retrieve a density field, and the resultant field is then added to the terrain via *CSG* operation. This works by rendering the object many times to a *slice* of a texture; the end result is similar to medical scan data, where multiple 2D images represent a 3D space. Rendering the slices is somewhat complex; it is similar to raytracing, where rays are fired through the object. However, significantly fewer rays are needed than when rendering a whole scene. Additionally, the rays begin inside the object and fire outward; they are considered inside if the number of intersections upon leaving the object is odd. *(E. Eisemann, X. Décoret, 2008)* Intersections with the object are represented by drawing a pixel onto a texture held in memory. By progressively drawing these pixels, we are left with a vertical or horizontal "slice" of the object on the texture. Repeating this process along the model results in many slices, and these slices can be used as input for the density field function, thereby completing the terrain-destruction-physics-terrain cycle.
There is, of course, a limitation to this process; for each of the objects that have fallen, a density field texture must be generated. This is likely to be memory-intensive, as for each successive terrain destruction operation, more textures must be generated.

This could severely limit the effectiveness of the method, but optimizations can be made. For example, instead of a texture for every object, the program could wait until groups of objects come to rest and perform the slice technique on multiple shapes at once time, reducing the number of textures needed at the expense of accuracy.

2.5 - Performance Analysis

This segment of the application is necessary for meaningful performance information to be collected.

Commonly, games are judged in performance by how many *Frames Per Second* are rendered. However, this only tells us an average of the time taken per frame over the course of the program's operation, and not specifically which parts of the program may or may not be causing performance issues. Instead, the application will be judged by calculating the time taken in milliseconds for each of the previous 3 segments to complete their given task with their current parameters. This will be done using a technique known as *instrumentation*. This technique involves inserting small blocks of code which act like stopwatches, timing how long other blocks of code take to perform their tasks. This is normally done by capturing the current system time, and then by comparing the system time again later on during the program's run cycle.

By recording the *average frame-time* of the software as a whole as well as these individual pieces, we can evaluate and judge the performance of each technique and each combination of techniques.

The application logs these times into files, which can then be parsed by applications like *gnuplot* or *MATLAB* to be graphed against eachother.

To make the information more relevant, information will be written into the files when certain events happen; for example, when the user performs an action which alters the terrain's makeup. That way, it is easier to perceive the cause and effect of certain actions upon the overall performance.

In addition to pure speed, an important consideration when dealing with applications is the *memory footprint* of the application. For example, as more physics objects are generated, each takes up space in memory. The more objects there are, the larger the memory footprint is. Since devices have finite memory in which to store objects, care must be taken not to introduce methods which unnecessarily use large regions of memory.

Tracking memory usage is a little more complex than tracking execution speed, but the additions to the profiling tools in *Visual Studio 2015* make this much easier, as it now provides memory usage graphs, snapshots and statistics. Memory snapshots allow us to explore the state of a program at any point in time, and so it is possible to identify exactly where and which objects are consuming large amounts of memory.

Using these techniques and data sets, each method will be examined first against its own parameters (e.g, grid size, texture resolution) to find the most performant selection on the test hardware. Then, the combinations of different methods will be analysed (using the most performant parameters for each method as a base), finding the most performant selection once again. Finally, all methods and combinations will be listed together from best performance rating to worst. Screenshots of the application will be provided alongside each of the combinations for a visual comparison of the changes.

Then, the most balanced selection (the median performance point) will be calculated from the results, and a resultant image generated for that case.

3. Results

This section will discuss the results of the performance analysis mentioned in the *Methodology* section.

Firstly, it should be noted that the application's performance results will be highly dependent on the computer hardware used for testing. As such, the following table provides system specification information for the results.

Table 1: System Specifications

CPU	Intel Core i5 2500K (3.3 GHz, 4 Cores, 64-bit)
GPU	Nvidia GTX 680 (4GB VRAM)
RAM	8GB DDR3 (1333 MHz) RAM
Display Size	1920x1080, Application At 1280x720
Operating System	Microsoft Windows 10

The overall performance of the application now will be examined. Performance is measured mainly in average frame-time; that is, how long the application takes to complete one cycle and draw a frame to the screen. Because the performance of the render and the performance of the application's internal updates (for physics calculations) are affected by different stimuli, they have been recorded separately from the overall time taken to display a single frame. These two measures are defined as *draw frame-time* and *update frame-time.* The overall frame-time is summed from these two. A method is considered to have better performance the lower the frame-time is, in individual parts and in overall. Additionally, the memory footprint of a method is also taken into account as a measure of performance.

Note: in this section, there are multiple graphs. Graphs will be shown on their own pages, as their resolution is somewhat large. Images of the program in operation are in Appendix A.

The first performance test was done to determine a baseline average performance for each of the terrain representation methods, measuring average frame-times and memory footprints. When measuring these averages, the application was run for 10 seconds.

3.1 - Grid-Based Performance

The application uses two methods to represent terrain, as discussed in the *Methodology;* the first of these, *Grid-Based Naive Marching Cubes*, relies on a 3D grid of points for terrain display. By increasing the resolution of this grid, a larger area of terrain can be displayed at once. As expected, increasing the grid resolution correlates to steady increases in frame-time in rendering (*Figure 16: Average Rendering Frame-Times for Grid-Based Naive Marching Cubes Parameters*). However, somewhat inconsistent increases can be seen in the physics-update frame-times (*Figure 17: Average Physics-Update Frame-Times for Grid-Based Naive Marching Cubes Parameters*), where there is an upward trend to frame-times but not necessarily a correlation, as increases in grid resolution do not necessarily impact the physics updates in the same way. Memory footprint, as expected, increases sharply as storing the information for the grid increases somewhat exponentially (*Figure 18: Average Memory Usage for Grid-Based Naive Marching Cubes Parameters*).

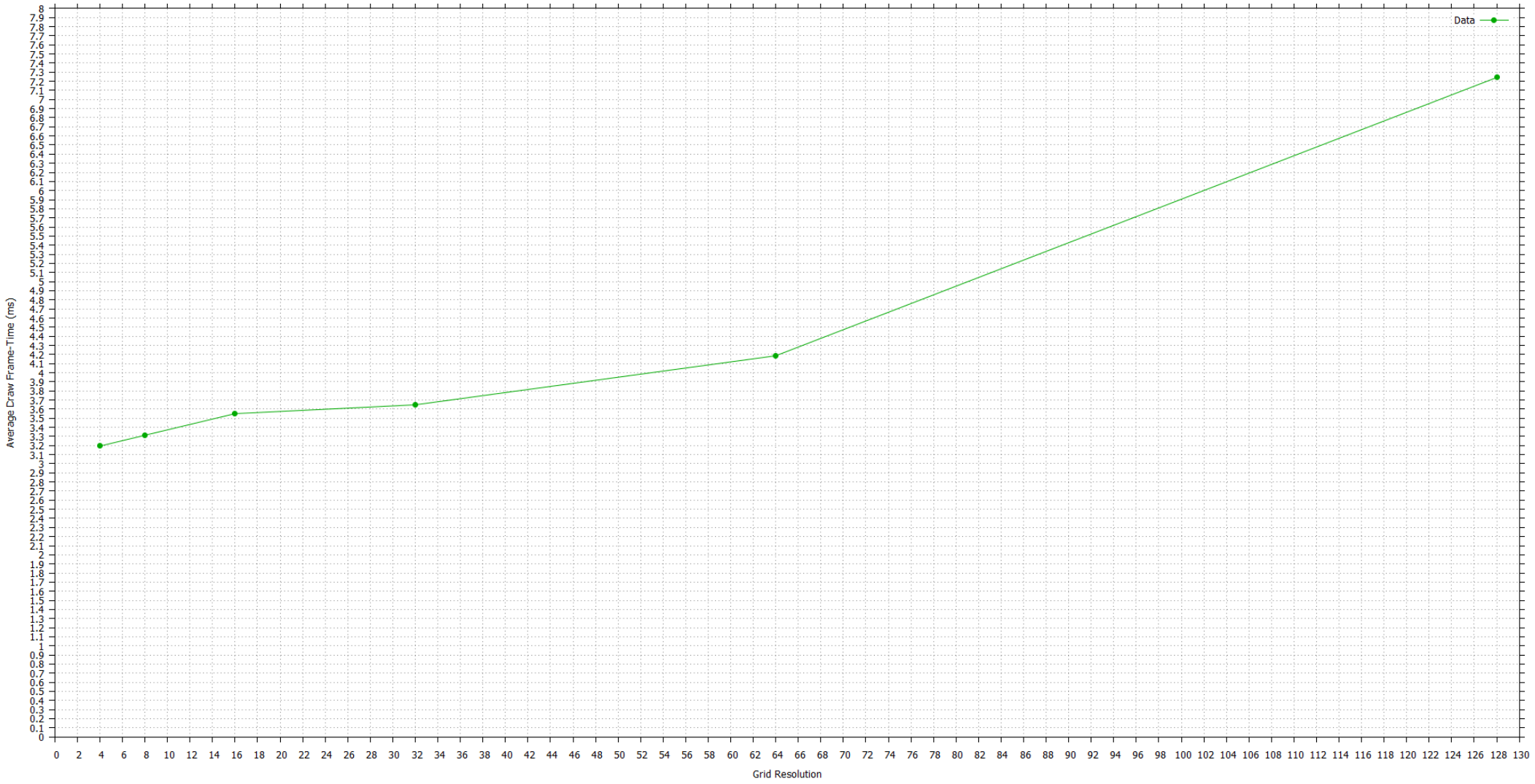


Figure 16: Average Rendering Frame-Times for Grid-Based Naive Marching Cubes Parameters

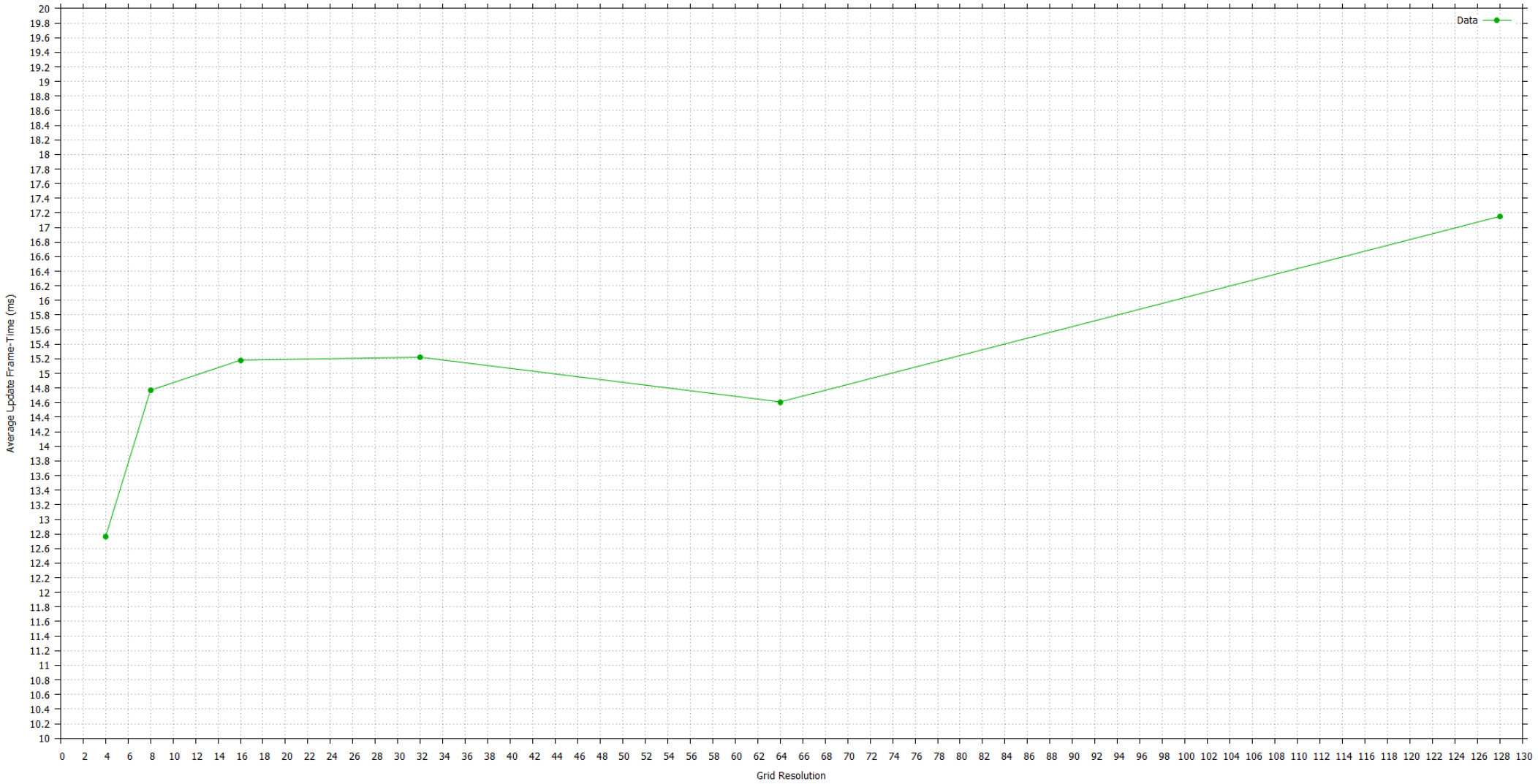


Figure 17: Average Physics-Update Frame-Times for Grid-Based Naive Marching Cubes Parameters

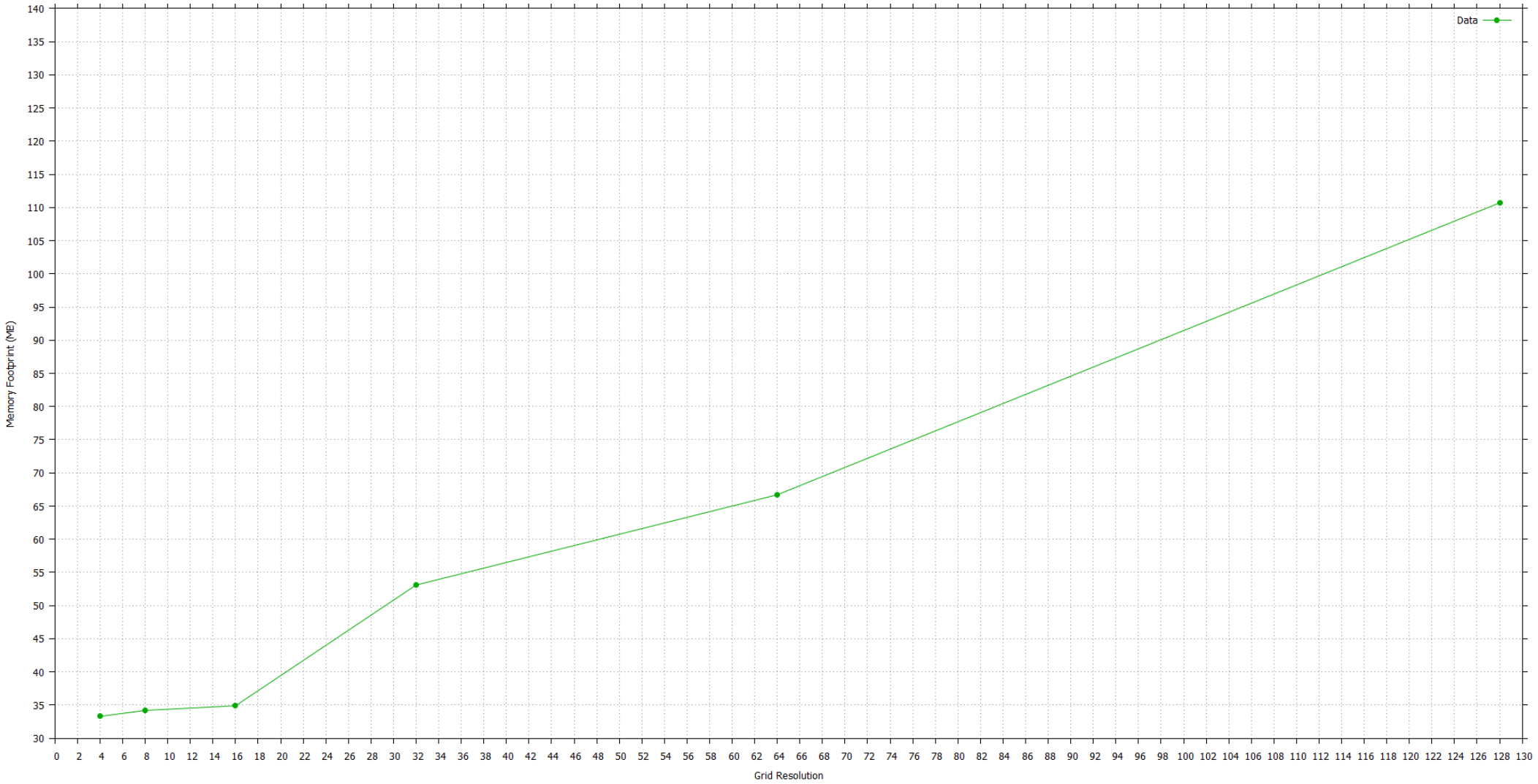


Figure 18: Average Memory Usage for Grid-Based Naive Marching Cubes Parameters

3.2 - Distance-Based Raymarching Performance

The same test was repeated for the second terrain representation method: *Distance-Based Raymarching.* The performance for this method is tied primarily to the resolution of the texture that the method writes to, as it must perform the raymarch calculations for every pixel of the texture. The test was performed by measuring performance against the percentage of total render resolution. The original full-size resolution was *1280 x 720,* so the smaller resolutions are relative to that size.

Predictably, increasing the rendering resolution decreases the render performance (*Figure 19: Average Rendering Frame-Time for Distance-Based Raymarching Parameters*). The same is true for both physics-update and memory footprint (*Figures 20, 21*). Importantly, however, the increases for physics-update and memory footprint are much, much smaller than with the *Grid-Based* method. This is because instead of storing a 3D grid of points and all the data associated with such, the software is merely holding a single 2D texture. Even more interesting is the revelation that the average rendering time for *Distance-Based Raymarching* is, even at full resolution, considerably more performant than *Grid-Based.* This is in conflict with the assumptions made during the construction of this project. Rather than *Distance-Based Raymarching* being more accurate yet more computationally expensive than *Grid-Based,* it would appear from these results that *Distance-Based Raymarching* is clearly both more performant and more accurate than *Grid-Based.* Visual quality is demonstrably higher, as a much larger quantity of terrain can be displayed without additional performance issues (*Figures 19, 20 & Figures 35, 36 in Appendix A).* However, the average frame-times while only displaying terrain do not provide enough information about the practical application of these techniques as terrain modification and reconstitution are applied.

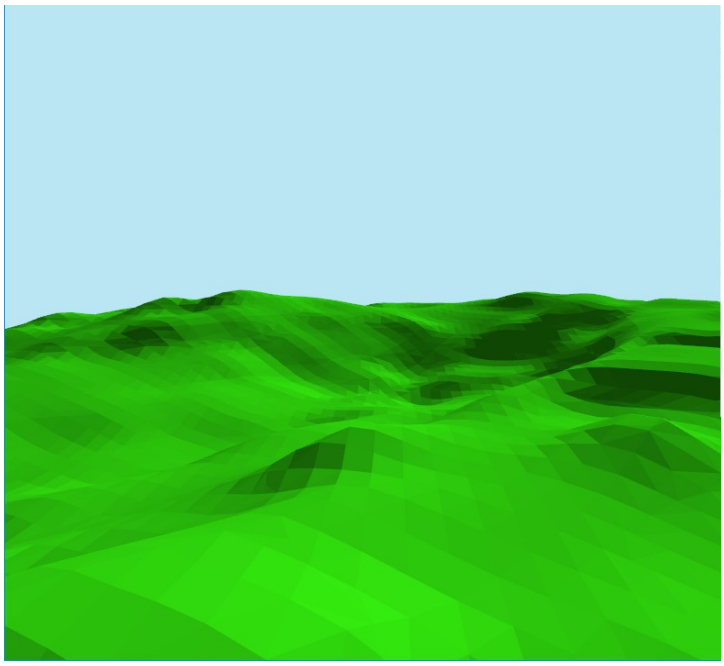


Figure 19: Application in Grid-Based Mode, at full grid resolution.

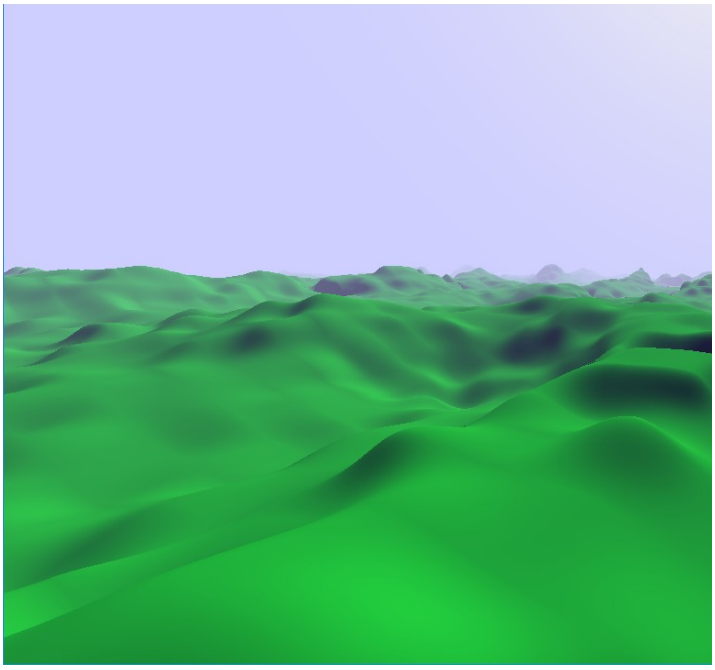


Figure 20: Application in Distance-Based Raymarching Mode, at full resolution.

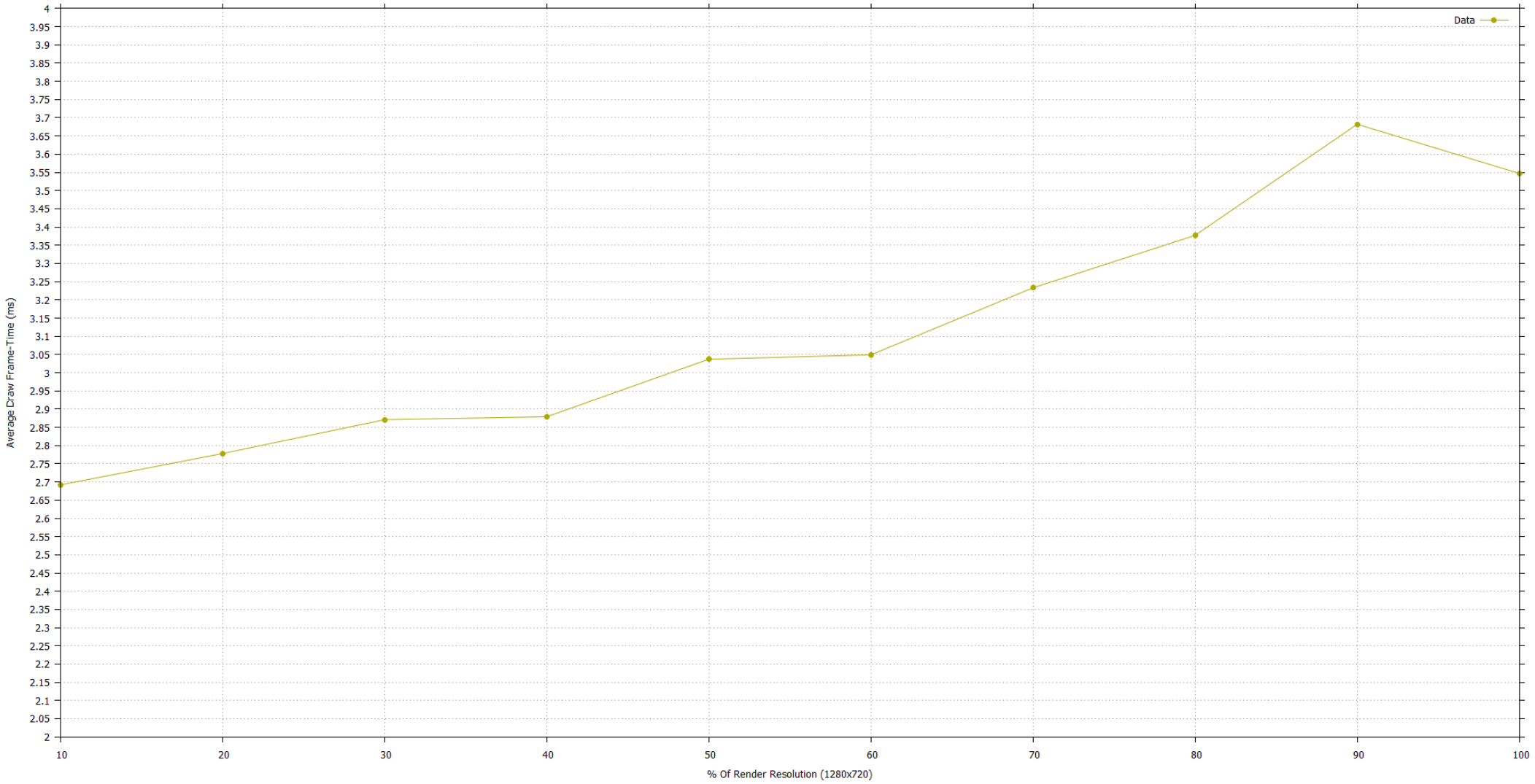


Figure 21: Average Rendering Frame-Times for Distance-Based Raymarching Parameters

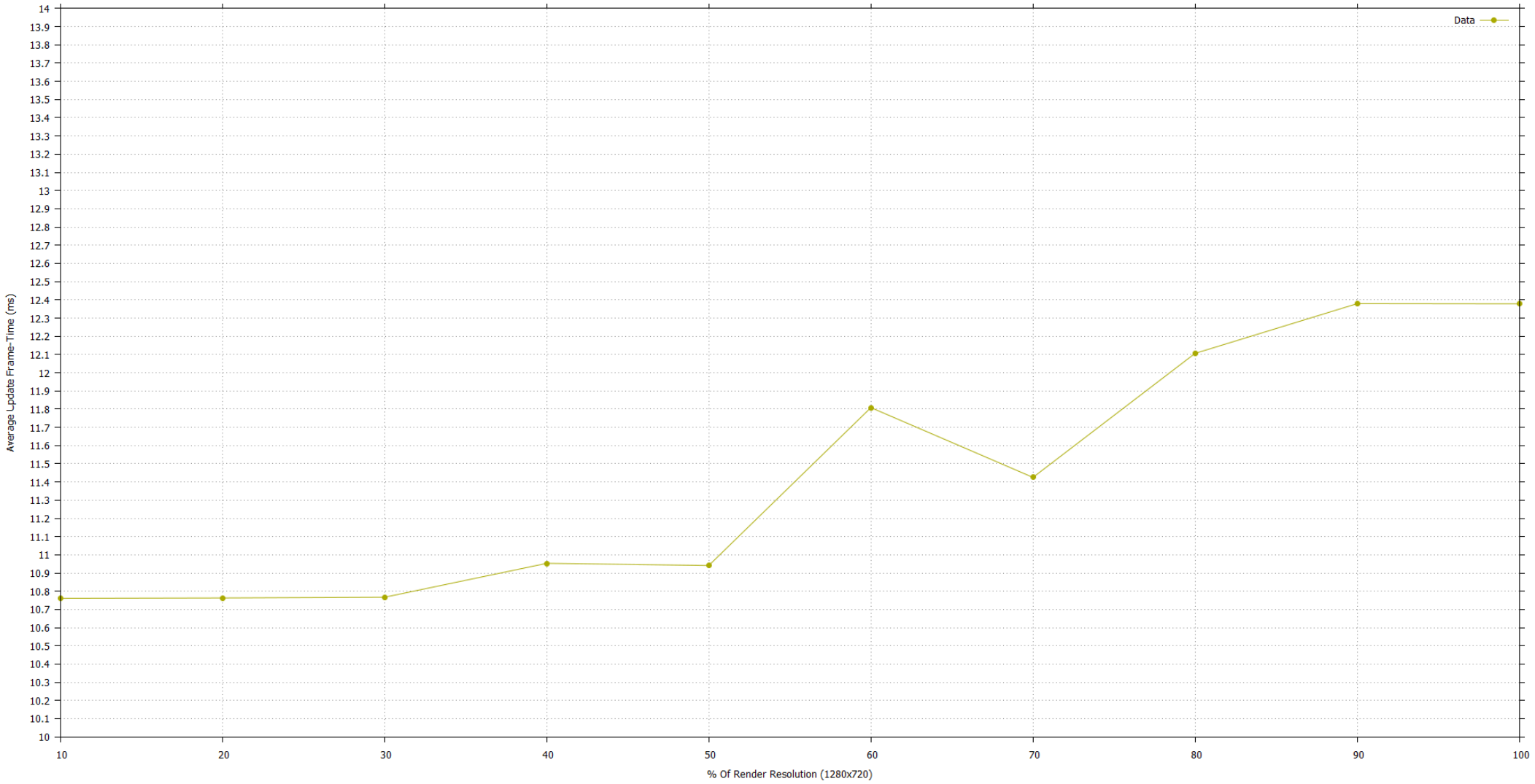


Figure 22: Average Physics-Update Frame-Times for Distance-Based Raymarching Parameters

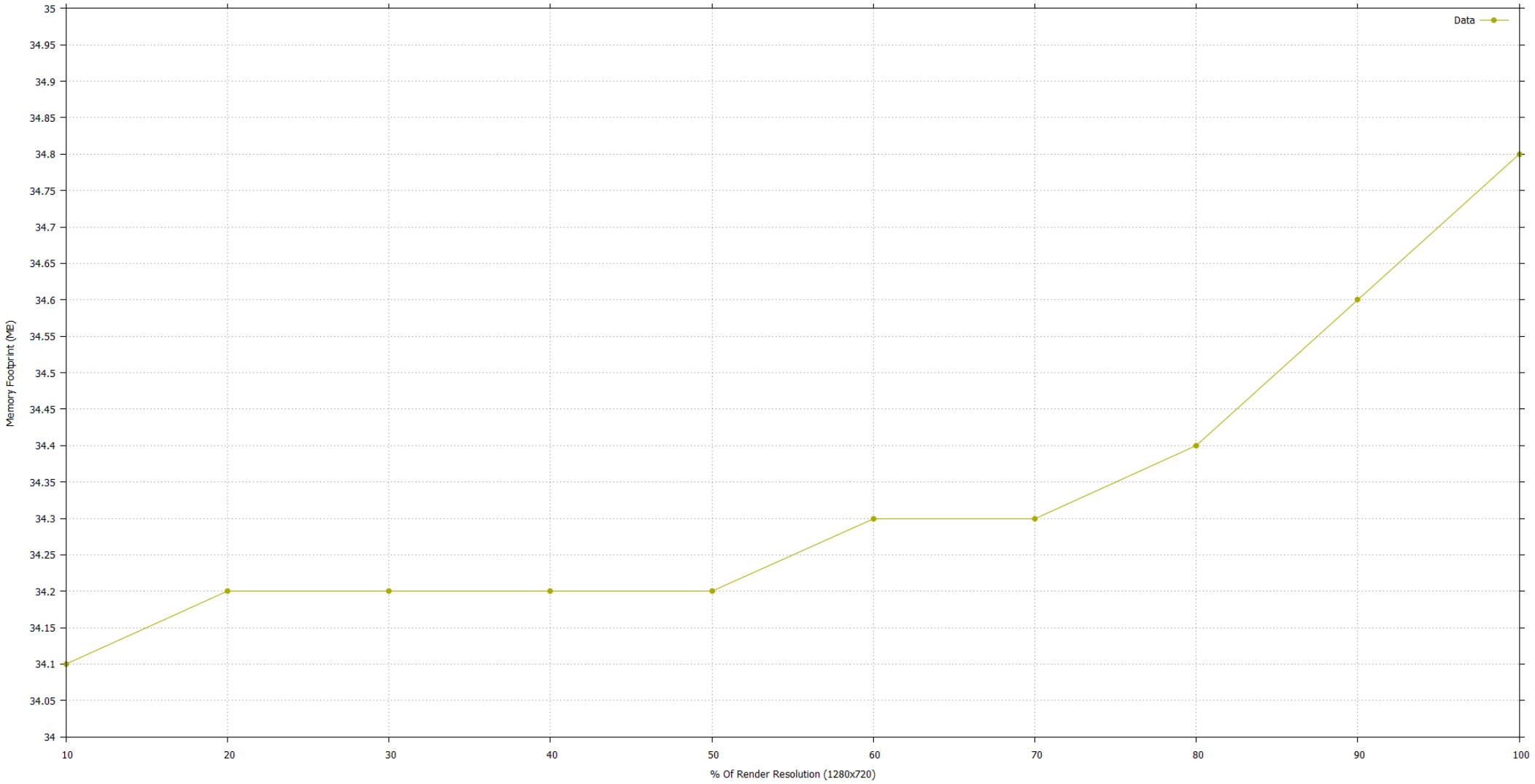


Figure 23: Average Memory Usage for Distance-Based Raymarching Parameters

3.3 – Event Logging & Overall Performance

To this end, a new test was required: for this test, frame-time was continually measured and logged by the software, with important events (such as user-interaction events like modifying terrain or events where physics fragments were reconstituted into terrain form) being logged and timestamped for viewing on performance/time graphs.

Events on graphs are marked as follows:

- Red rectangles are moments where physics recalculations took place.
- Yellow rectangles are moments where the user initiated a change to the terrain.
- Blue or purple rectangles are moments when the user switched terrain rendering methods.
- Green rectangles are moments when active physics objects merged back into the terrain.

With these results, ignoring spikes where physics information was recalculated for the *Raymarching* implementation, it can be observed (*Figure 24*) that as continuous changes are made to the terrain (repeated carving operations, in this case) the *Raymarching* method begins to suffer performance degradation fairly quickly. Switching display mode to *Grid-Based (Figure 25)* during the program's operation reveals that while *Grid-Based* may have a lower average performance, it is capable of dealing with cumulative terrain changes more efficiently.

Physics recalculations (where the triangulated terrain is sent to the physics engine to be used for simulation), show a sudden spike in frame-time for renders but not for physics-updates (*Figures 26, 27).* This is because the physics information is collected by the GPU and transmitted back to the CPU; code is in place that prevents this from happening until the scene has been moved by an appropriate amount, so that performance is not hamstrung by this potentially expensive operation. If the program were to have its physics-updates and render updates decoupled and run on separate processing cores (a practice known as *multithreading)* then it is possible that the physics data could be calculated on the CPU-side instead; with both types of update operating in parallel, the user would likely

not notice any significant performance drop. Currently, the physics recalculations can be noticed as small “hiccups” in the program's otherwise smooth operation.

When segments of terrain are fractured into many physics objects, it can be observed that the performance worsens while these pieces are simulated by the physics engine (*Figure 28).* However, due to the fact that pieces are reconstituted back into terrain, the performance impact of those objects falls back towards the baseline. This is proof that converting physics objects back to density-field terrain is a highly performant way to physically simulate terrain destruction. In effect, this shows that while the specifics of the implementations may have their positive and negative sides, the overall method is sound. From this, we can draw our conclusions.

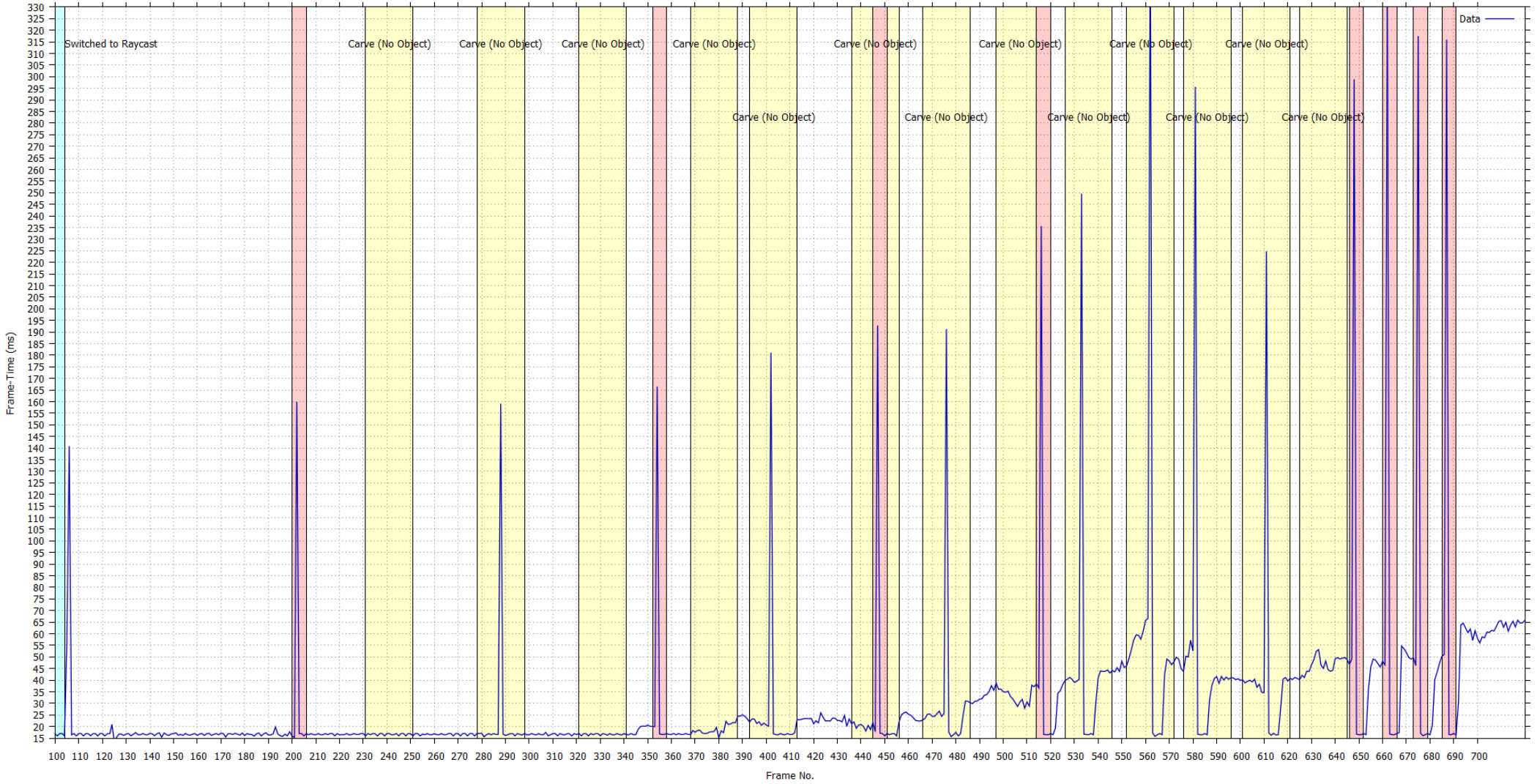


Figure 24: Gradual Performance Degradation after Repeated Carve Operations in Raymarch mode.

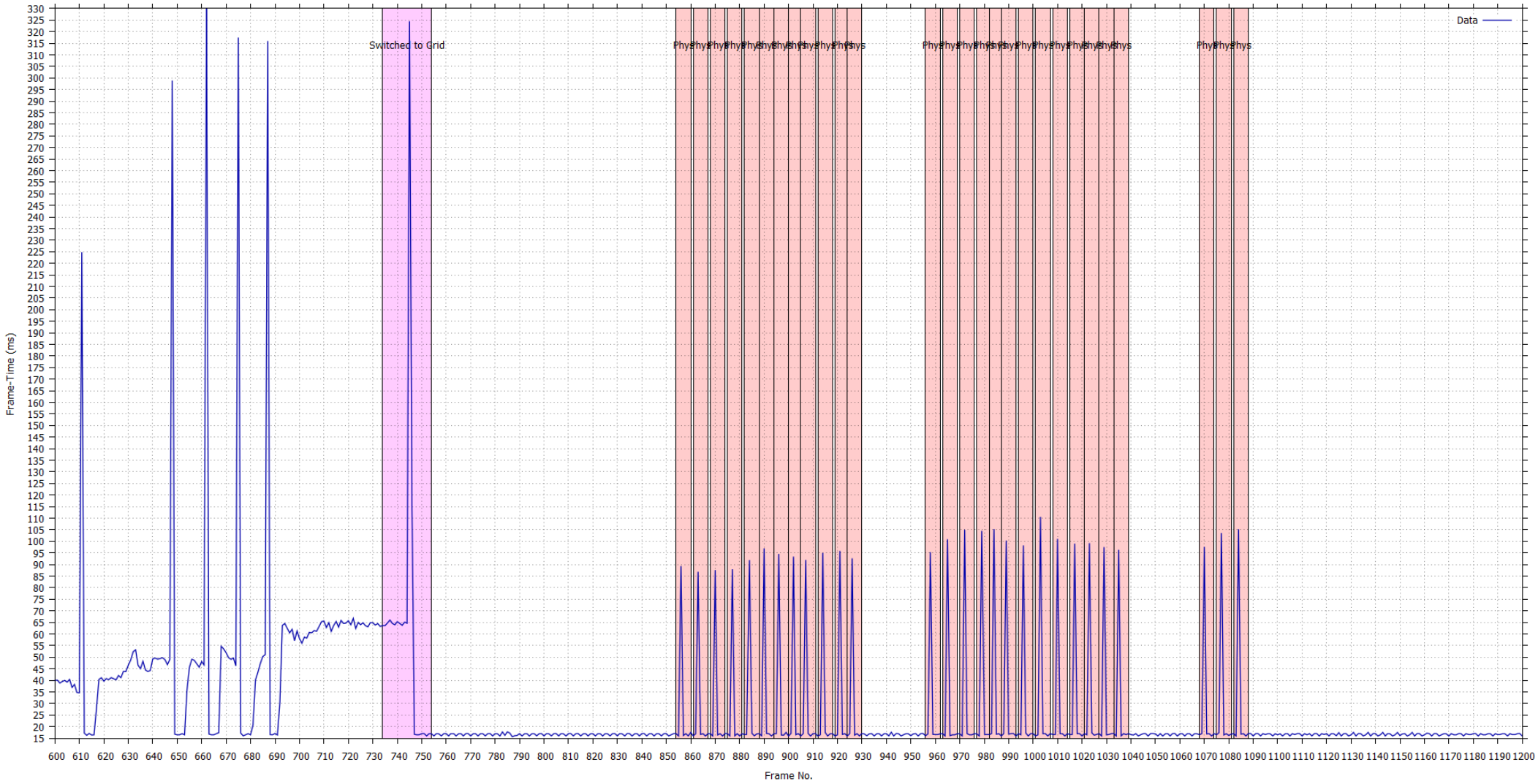


Figure 25: Performance returns to baseline after switching to Grid-Based mode.

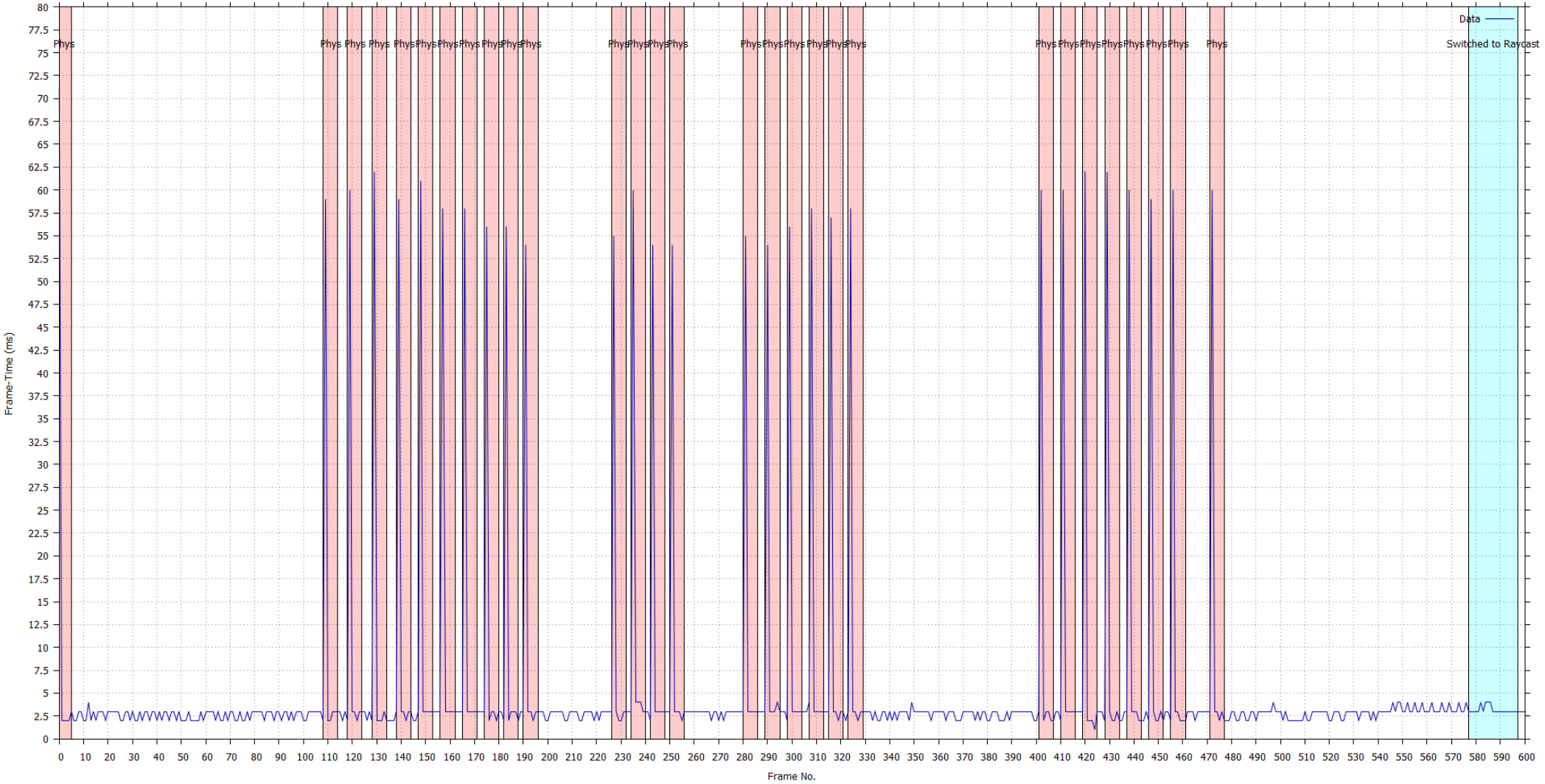


Figure 26: Performance Spikes in Render Frame-Times when Recalculating Physics Mesh

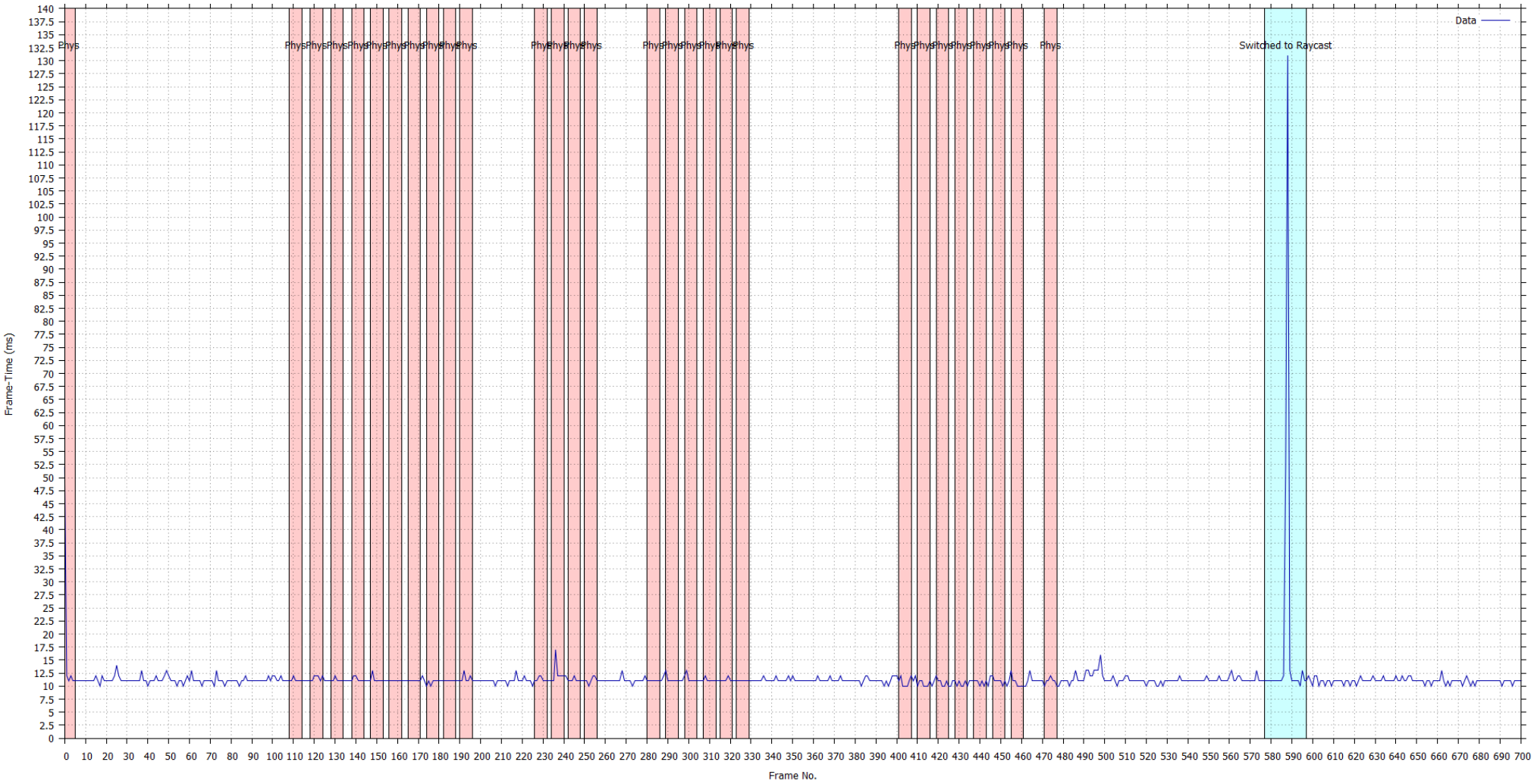


Figure 27: Absence of spikes in Physics Update Frame-Times when Recalculating Physics Mesh

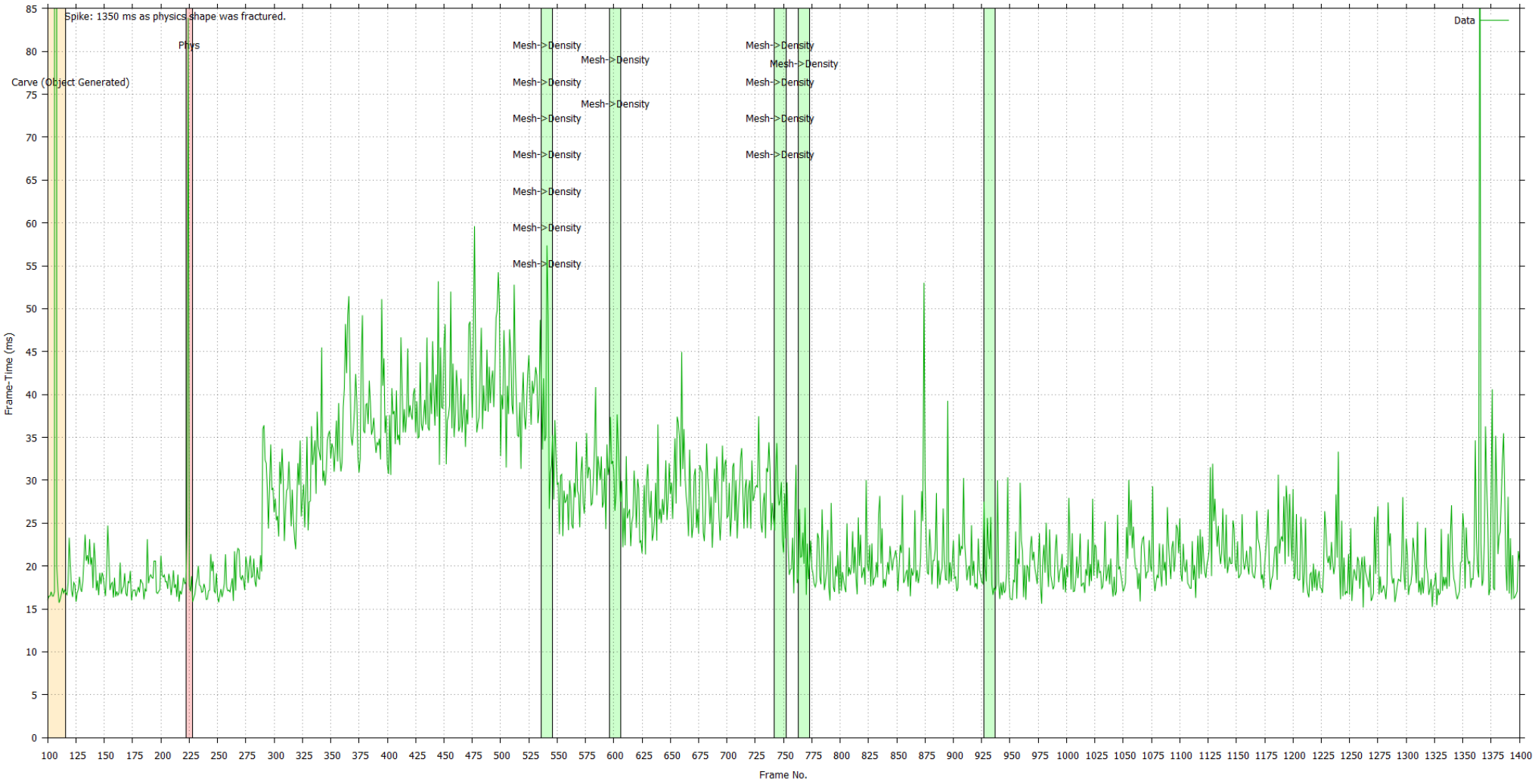


Figure 28: Performance impact reduced to baseline after objects re-combine with terrain.

4. Discussion & Conclusion

This section will contain a critical evaluation of the project's implementation, a discussion of future work that could be done in order to more fully explore the topic and answer the research question, and the conclusion of the project's findings.

4.1 - Critical Evaluation

The results discussed in the previous section outline the utility of these newer terrain representation techniques when dealing with real-time physics-based destruction. As such, the results provide an answer to the research question posed earlier, but this may not be an entirely accurate answer. The question asks which the “most efficient” method is; the application does not implement alternative methods for physics object fracture or re-combination, focusing mainly on terrain representation and storage instead; a conclusion therefore can be drawn based on the terrain representation's methods, with the other two components of the application performing a necessary task but being yet limited in their potential exploration.

Due to time constraints and the unforeseen complexity of these alternative methods, they were cut from the project in the interests of preserving the overall goal; had they not been cut, it is likely that the application would not be complete enough for analysis of the terrain-management components, and would have put the entire project at risk. Terrain re-composition is in a less complete state than physics object fracture; while the physics object fracturing does implement one of the discussed methods, the terrain re-composition itself merely consists of generating a sphere of terrain at the position of one of the terrain fragments and does not accurately represent the chunk of terrain at all. This extremely simplistic method was developed in order to prove the theory that reducing physically-simulated elements back into terrain would reduce the overall processing overhead in a real-time environment, without delaying development times further with the implementation of a more realistic method.

However, the current implementation does provide a good approximation of the overall techniques and their viability; indeed, in game environments, this simplification can be preferable depending on the nature of the game's aesthetic.

4.2 - Future Work

With additional time, the slight shortcomings could be fixed; the modular structure of the application allows for additional methods to be written relatively simply. Implementing the more complex re-composition method and additional fracturing methods would provide further data for analysis, and could lead to more detailed and concrete proof for, or indeed against, the central point of the project.

Given that the end-result of the application resulted in some qualitative elements, such as the visual accuracy or fidelity of the final product (as in a game-like environment), testing could have included a user-experience test with questionnaires. While qualitative research would not have informed the project's overall research question, the information from it would have allowed comparisons between methods to be made on more than a purely performance-oriented basis. The most performant solution is unlikely to be the best-case for every game development project, as the visual style or target platforms may be incompatible with certain technologies. Future exploration of the topic would almost certainly include qualitative surveys.

Additional future work would likely include a greater range of test hardware; using a single, high-performance platform like a desktop computer can provide basic information about the overall concepts of the project, but lacks the information necessary to judge which specific methods and combinations should be used on lower-performance platforms such as mobile devices.

4.3 - Conclusion

In conclusion, the project was a success, but has some minor issues which likely necessitate further research. The results indicate that using *Distance-Based Raymarching* in combination with a simple fracturing and recombination method is the most efficient way to simulate real-time, physics based destruction with scalar field terrains. This differed from the expectation that *Grid-Based Naive Marching Cubes* would be a more efficient but less accurate method. This revelation provides valuable insight into the nature of these kinds of terrains, and which kinds of processing are most efficient on the GPU in a real-time environment. With technology like this proving to be a viable option, more detailed and more interesting types of terrain can be modeled and interacted with in real-time in applications ranging from games and film production to seismic simulation environments.

References

Minecraft. 2009. [Computer Game]. PC. Mojang.

Space Engineers. 2013. [Computer Game]. PC. Keen Software House.

Red Faction. 2001. [Computer Game]. PC. Volition, THQ.

SIGGRAPH. 2015. [online]. Available from: http://www.siggraph.org/about/about-acm-siggraph
[Accessed 26 November 2015]

W. Lorsensen, H. Cline. 1987. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm.* [online]. Available from:
http://www.eecs.berkeley.edu/~jrs/meshpapers/LorensenCline.pdf
[Accessed 19 January 2016]

P. Bourke. 1994. *Polygonising a Scalar Field.* [online]. Available from:
http://paulbourke.net/geometry/polygonise/
[Accessed 20 January 2016]

Tao Ju et al. 2002. *Dual Contouring of Hermite Data*. [online]. Available from:
http://www.cs.wustl.edu/~taoju/research/dualContour.pdf
[Accessed 20 January 2016]

Quilez, I. 2008. *Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes.* [online]. Available from: http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf [Accessed 23 January 2016].

Gustafsson, D. 2014. *Cracking Destruction.* [online]. Available from: http://tuxedolabs.blogspot.co.uk/2014/05/cracking-destruction_13.html [Accessed 24 January 2016].

Sara C. Schvartzman, et al. 2014. *Physics-Aware Voronoi Fracture with Example-Based Acceleration.* [online]. Available from: http://jcgt.org/published/0003/03/03/paper.pdf [Accessed 27 January 2016].

O. Fryazinov, A. Pasko, V. Adzhiev. *BSP-fields: An Exact Representation of Polygonal Objects by Differentiable Scalar Fields Based on Binary Space Partitioning.* University of Bournemouth. [online]. Available from: http://eprints.bournemouth.ac.uk/18561/1/bsp_preprint.pdf [Accessed 1 February 2016].

E. Eisemann, X. Décoret. 2008. *Single-Pass GPU Solid Voxelization for Real-Time Applications.* [online]. Available from: http://maverick.inria.fr/Publications/2008/ED08a/solidvoxelizationAuthorVersion.pdf [Accessed 1 February 2016].

OpenFrameworks, 2015. *About OpenFrameworks.* [online]. Available from: http://openframeworks.cc/about/ [Accessed 8 November 2015].

voro++, 2016. *Overview.* [online]. Available from: http://math.lbl.gov/voro++/
[Accessed 13 February 2016].

Mojang. November, 2015. "*Minecraft Snapshot 15W46A".* [online image]. Available from:
https://mojang.com/2015/11/minecraft-snapshot-15w46a/
[Accessed 12 April 2016].

Keen Software House. 2016. *Untitled Image.* [online image]. Available from:
http://www.spaceengineersgame.com/pictures.html
[Accessed 12 April 2016].

March. 2011. "*massive floating mountains"* [online image]. Available from:
http://www.minecraftseeds.info/2011/03/9028489474908844496.html
[Accessed 23 February 2016].

Ronen Tzur. February 2004. *Untitled Image 1.* [online image]. Available from:
http://www.oocities.org/tzukkers/isosurf/isosurfaces.html
[Accessed 24 February 2016].

Ronen Tzur. February 2004. *Untitled Image 2.* [online image]. Available from:
http://www.oocities.org/tzukkers/isosurf/isosurfaces.html
[Accessed 24 February 2016].

Ronen Tzur. February 2004. *Untitled Image 3.* [online image]. Available from:
http://www.oocities.org/tzukkers/isosurf/isosurfaces.html
[Accessed 24 February 2016].

Ronen Tzur. February 2004. *Untitled Image 4.* [online image]. Available from: http://www.oocities.org/tzukkers/isosurf/isosurfaces.html [Accessed 24 February 2016].

Iñigo Quilez. 2008. *"slisesix".* [online image]. Available from: http://iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm [Accessed 26 February 2016].

Sara C. Schvartzman et al. 2014. *"bunnyLearn.png".* [online image]. Available from: http://www.gmrv.es/Publications/2014/SO14a/ [Accessed 3 March 2016].

Red Faction Wikia. 2009. *"227129-geomod super.jpg".* [online image]. Available from: http://redfaction.wikia.com/wiki/File:227129-geomod_super.jpg [Accessed 3 March 2016].

Anderson, B. 2016. *Untitled Image.* [online image]. Available from: http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html [Accessed 3 March 2016].

Esteve, J. 2011. *"Fig 1. Voronoi cell intersecting source geometry to form fragment.".* [online image]. Available from: http://www.joesfer.com/?p=69 [Accessed 16 March 2016].

Esteve, J. 2011. *"Fig 2. Slicing the geometry with the faces of a VD cell"*. [online image]. Available from: http://www.joesfer.com/?p=69 [Accessed 16 March 2016].

Bibliography

Nguyen, H. 2007. *GPU Gems 3* (first ed.). Addison-Wesley Professional.

Quilez, I. 2008. *Modeling with distance functions.* [online]. Available from: http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm [Accessed 11 February 2016].

Bertout, C & Schneider, P. 2005. *Introducing structured abstracts for A&A articles.* Astronomy & Astrophysics. Vol. 441. pp E3-E6. October 1, 2005.

Trettner, P. 2013. *Terrain Engine Part 1 - Dual Contouring.* [online]. Available from: https://upvoid.com/devblog/2013/05/terrain-engine-part-1-dual-contouring/ [Accessed 19 January 2016].

Lengyel, E. 2010. *Voxel-Based Terrain for Real-Time Virtual Simulations.* PhD diss, University of California.

Eberly, D. 2008. *Clipping a Mesh against a Plane.* [online]. Available from: http://www.geometrictools.com/Documentation/ClipMesh.pdf [Accessed 12 March 2016].

Cepero, M. 2010. *From Voxels to Polygons.* [online]. Available from: http://procworld.blogspot.co.uk/2010/11/from-voxels-to-polygons.html [Accessed 19 January 2016].

Casella, T. 2013. *Artist-Driven Fracturing of Polyhedral Surface Meshes.* [online]. Available from: http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=2220&context=theses [Accessed 24 February 2016].

Kynd. 2015. *Using GLSL Transform Feedback in OF 0.9.0.* [online]. Available from: http://www.kynd.info/log/?p=632 [Accessed 18 February 2016].

APPENDIX A: Application Images

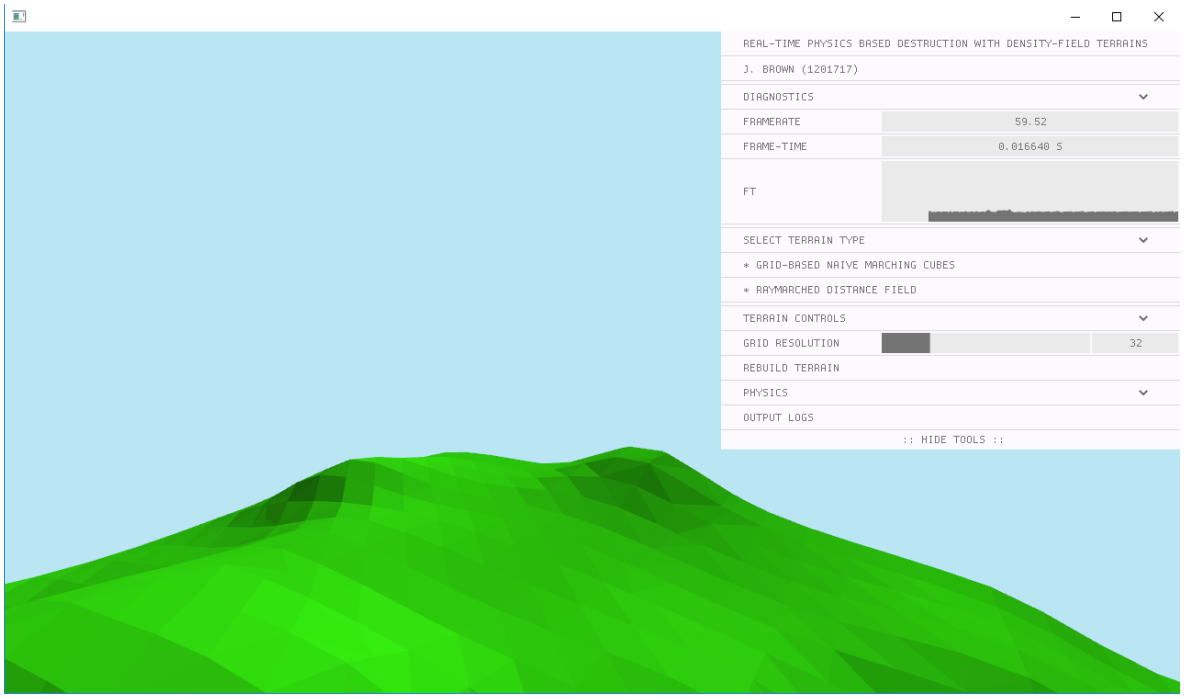


Figure 29: Project Application, Grid Mode, Low Grid Resolution.

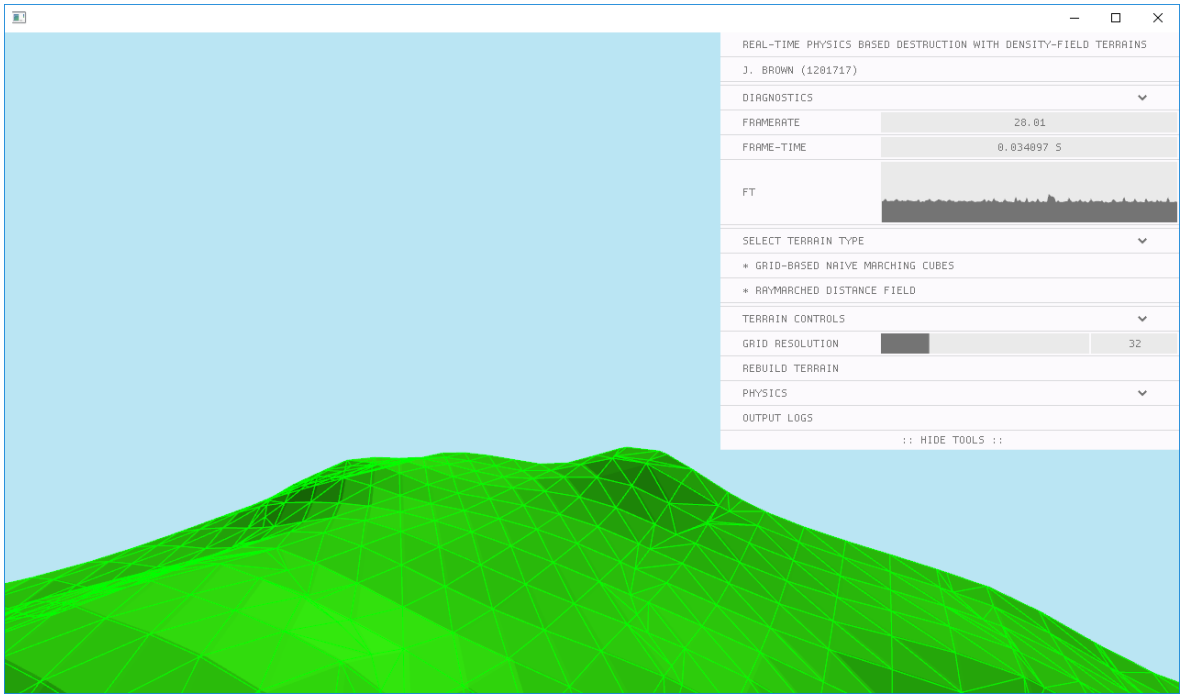


Figure 30: Project Application, Grid Mode, Low Grid Resolution with Physics Overlay.

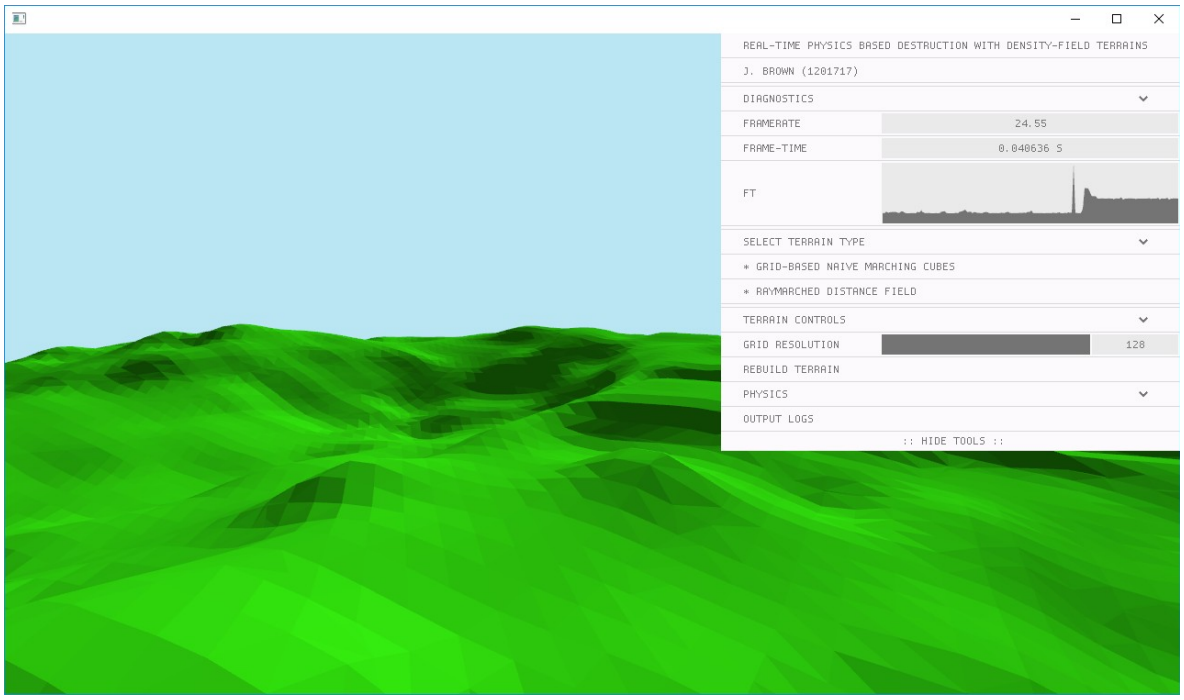


Figure 31: Project Application, Grid Mode, High Grid Resolution.

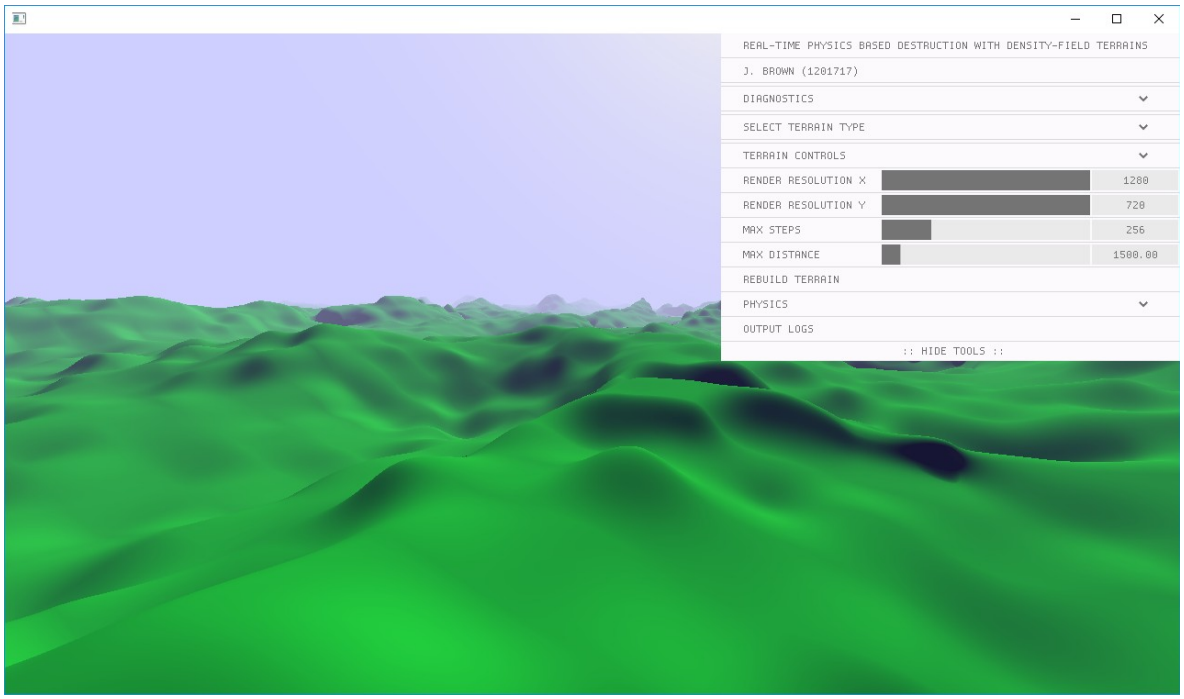


Figure 32: Project Application, Raymarching Mode, Full Resolution.

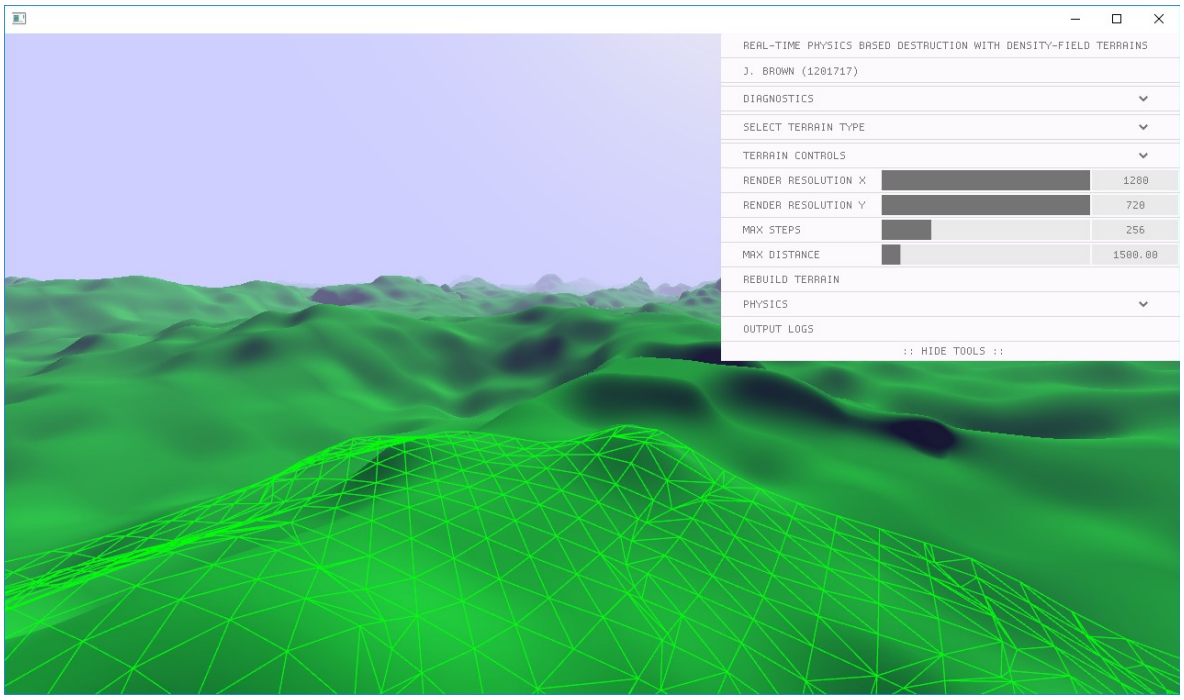


Figure 33: Project Application, Raymarching Mode, Full Resolution with Physics Overlay.

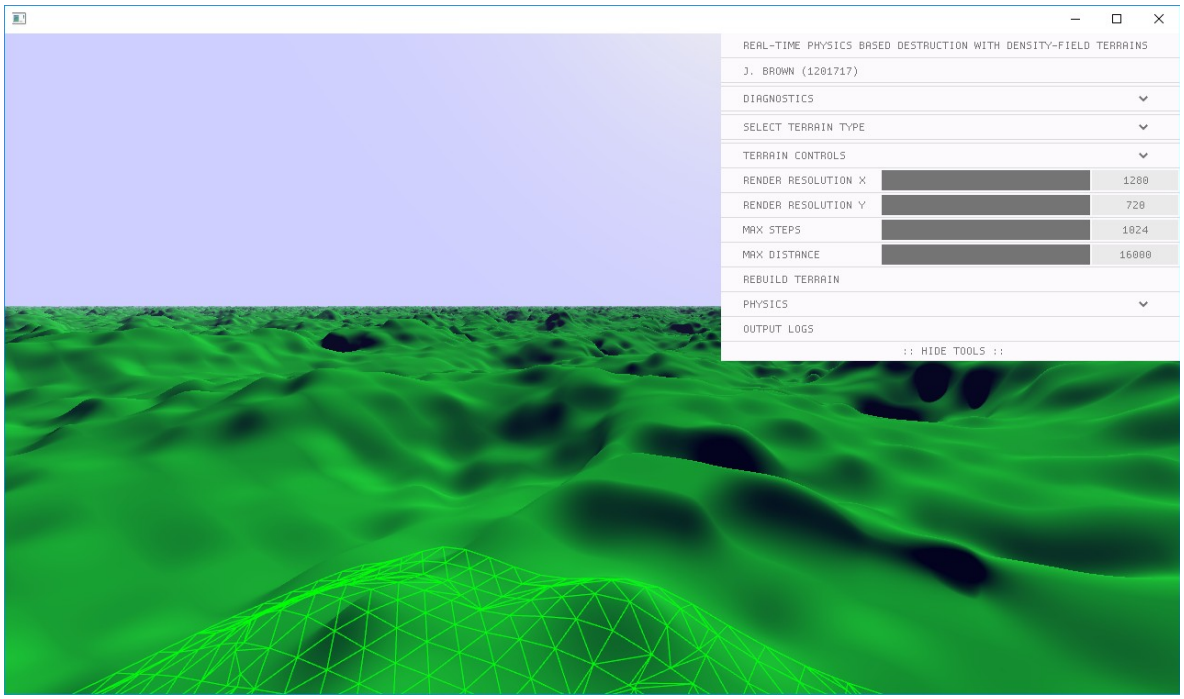


Figure 34: Project Application, Raymarching Mode, Full Resolution with Physics Overlay and Render Distance Increased to Maximum

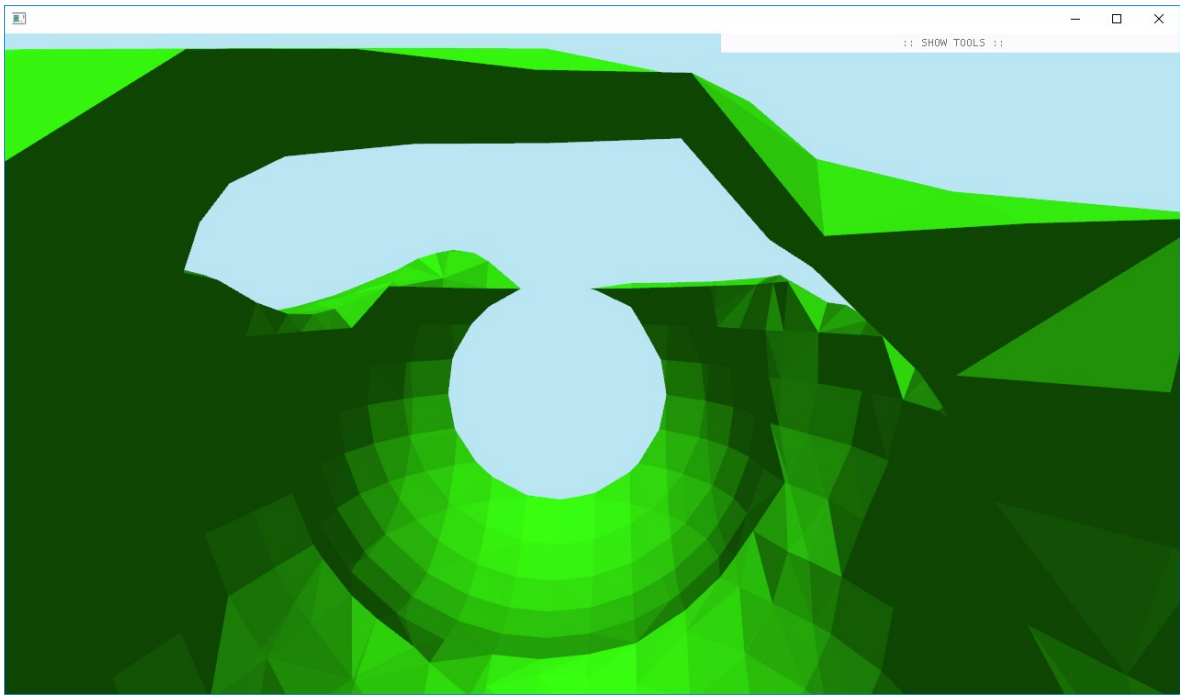


Figure 35: Project Application, Grid Mode, Tunnel in Terrain.

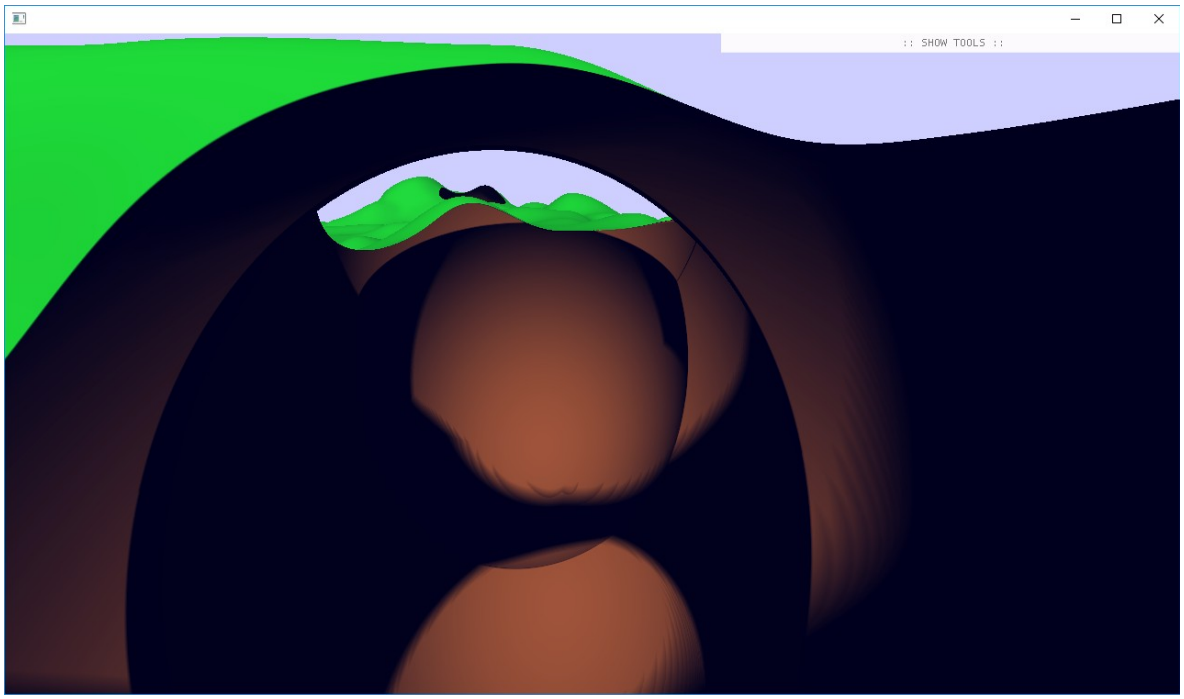


Figure 36: Project Application, Raymarching Mode, Tunnel in Terrain.